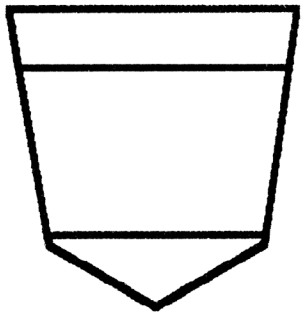
MACHINE LANGUAGE PROGRAMMING

SHARP PC-1500 & RADIO SHACK PC-2 POCKET COMPUTERS



© Copyright 1983 POCKET COMPUTER NEWSLETTER

MACHINE LANGUAGE PROGRAMMING
THE SHARP PC-1500
AND RADIO SHACK PC-2
POCKET COMPUTERS

Hidden beneath the surface of pocket computers such as the Sharp PC-1500 and Radio Shack PC-2 models is a secret world of unique and powerful capabilities. Alas, this world is only open to the initiated who have the willingness and stamina to study and learn *machine language programming!*

Machine language programming (or MLP as it will frequently be referred to here) is really what makes the entire system work. Indeed, the BASIC language itself and all the capabilities of the pocket computer are programmed in this most fundamental language of the microprocessor.

MLP is a complex subject. The reason computer languages such as BASIC were developed is because the average user of computers would like to be able to use them easily. Such people do not want to be bogged down in a myriad of technical details about how the computer itself operates. Details that have little to do with solving the problem at hand. They want to be able to quickly express to the computer how to perform a specific set of operations in order to obtain desired answers. The trade-off they make (frequently without knowing it) is one of speed of programming versus speed of computer operation. The speed of a computer, compared to that of the human mind, is vastly superior. Allowing it to take hundreds or even thousands of times longer than its most efficient arrangement would require is often unnoticeable to the casual user. Thus, the average user has little regard for the fact that the PC may take a thousand times longer to execute a program written in BASIC than it does to perform the same procedure when written in machine language.

When one programs in machine language one has to be concerned with a lot of technical details. These details often have little to do with solving the overall problem for which the computer is being used. If, for example, one has to perhaps deal with 20 or 30 parameters when programming a computer using a language such as BASIC, one might have to deal with 500 to 1000 pieces of information (instructions, storage locations, sequences, procedures, etc.) in order to successfully program the same problem using machine language. The reward for being able to do this, however, might well be an overall increase in the speed at which a specific problem could be solved. This improvement in the rate at which a specific task might be accomplished can often be measured in orders of magnitude, such as ten, one hundred or a thousand of times faster!

Access to the computer at the machine level can also open a whole new world of opportunities. This is especially true if one wants to deal at the hardware level in such areas as using the PC to perform real-time control operations. Generally these kinds of specialized applications cannot be successfully approached using high level languages, such as BASIC, because of speed or physical interfacing restraints.

A Piece at a Time

As complex as machine language programming can be, it is possible to make the chore of learning how to program in machine language be relatively easy to bear. The trick lies in breaking down the operation of a computer into its simplest parts. This makes each section easier to understand. Eventually the pieces can be placed back together as one gains familiarity with the fundamental aspects.

How simple can we get? How about defining a computer as consisting of two essential divisions: (1) a CPU and (2) memory.

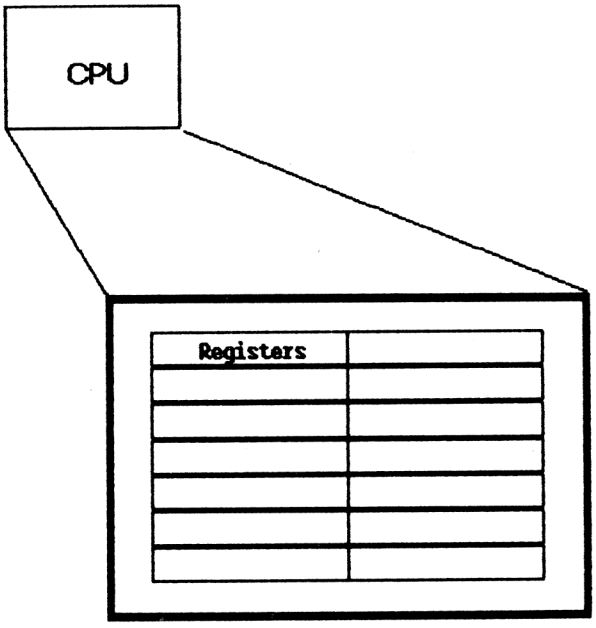
The CPU (Central Processing Unit) in a pocket computer is a microprocessor. That is, it is an integrated circuit that is the heart of the system. In the Sharp PC-1500 the CPU integrated circuit is referred to as an LH5801. It is a proprietary device made especially for that pocket computer. (Since the Radio Shack PC-2 is simply a custom-made version of the Sharp PC-1500, unless otherwise noted in this text, operation of the PC-2 can assumed to be the same.) The nomenclature LH5801 has no special significance to anyone other than the manufacturer. It is essentially nothing more than a part number to identify that particular type of electronic device.

A CPU such as the LH5801 is able to do a number of essential operations. But, basically what is does is transfer information to and from memory. Part of what it transfers is its own instructions. That is, the directives that tell it what to do!

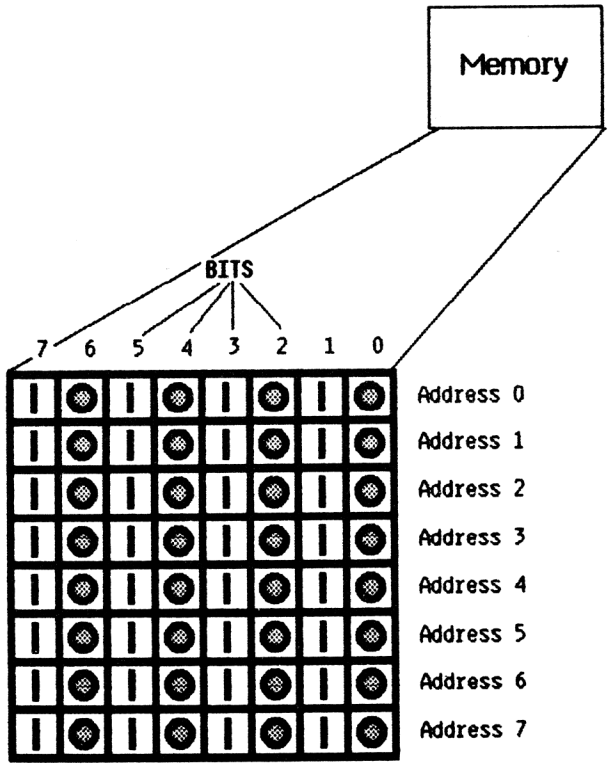
The memory elements in a computer hold two basic types of information. Instructions that tell the CPU what to do and data that is used to solve problems. To the uninitiated it is impossible to tell which parts of memory do what. You will soon become among the initiated.

A Step at a Time

A CPU essentially consists of a number of registers, some electronic paths that interconnect these registers, and an array of logic-performing circuits that control what happens within and between the registers.



Memory essentially consists of an orderly arrangement of registers and a method of addressing or selecting each register. In theory there could be up to 65,536 registers (64K) in the main memory bank of a PC-1500, each register of which would have a unique address in the range 0 - 65,535. (Because of the manner in which electronic circuits operate, numbering of items such as registers and cells within computers always start with zero.) Memory registers in a PC-2 have eight cells. Each cell can assume the binary state of one or zero. With eight cells making up a register, 256 different binary patterns or values (ranging from 0 through 255) can be stored in each memory register.



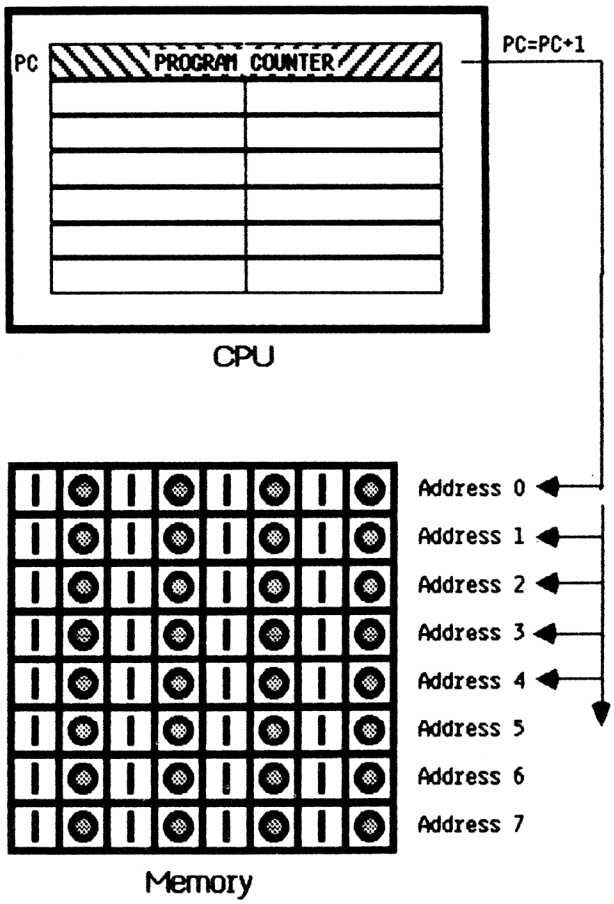
The Program Counter

The first thing that has to happen in order for the CPU to communicate with memory is for it (the CPU) to have an "address" specifying a particular memory register with which it is to transact its business. Would you believe that there is a special register within the CPU whose sole task is to keep track of where the CPU should obtain its next "communications" or instruction? Yep! And it has a very apt name: the *program counter.* The program counter in a PC-1500 is 16 bits wide. Now it just so happens that a register of 16 binary cells can represent up to 65,536 patterns -- in the range 0 through 65,535 -- which (coincidently, eh?) turns out to be the maximum number of memory registers that a pocket computer of the type being discussed can theoretically have in its primary memory banks. Thus, the program counter can in principle hold all the values that could translate into valid memory register addresses.

Whenever the CPU needs to obtain a instruction or an instruction-related piece of

 Copyright 1983 POCKET COMPUTER NEWSLETTER

information from memory, it looks at the contents of the program counter to ascertain the appropriate memory address.

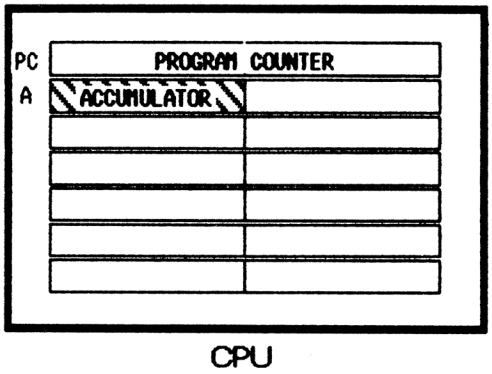


Once started, the operation of the program counter is essentially automatic. Whenever an instruction is "fetched" from a memory location, the value contained in the program counter is automatically incremented to "point" to the next memory address. There are, however, a few exceptions to this automatic incrementing sequence. A few special classes of directives can alter the contents of the program counter. Can you guess these classes? That is hardly a fair question at this point, but as you will eventually learn in detail, the classes of instructions known as "jumps" and "calls" can change the contents of the program counter.

How does a computer such as the PC-1500 get started? Well, one of the first things that happens when power is applied to the unit is that the program counter is set to contain the address in memory where the CPU is to begin finding machine language instructions!

The Accumulator

There is an 8-bit register in the CPU that can sort of be considered as a "jack-of-all-trades." This register is able to hold information while the CPU is in the process of performing other operations -- such as fetching another instruction from memory. It also works in conjunction with other internal CPU registers such as the arithmetic and logic unit (ALU). In this regards it is capable of performing mathematical operations such as addition, subtraction and Boolean logic (OR, NOT, AND, etc.). When a series of calculations are being performed this register can directly accumulate intermediate results. Hence the derivation of its name: *accumulator.*



One of the most frequent uses of the accumulator (which will frequently be abbreviated as the A register in this text) is simply to serve as a "scratch pad." That is, it is simply used to hold a number or binary pattern while the CPU obtains another operator. The accumulator in the PC-1500 can be loaded with a value obtained from memory or from another CPU register.

When directed to do so, the value in the accumulator can be added or subtracted from the value of another CPU register or a location in memory. It can also perform logic operations with the contents of other CPU registers or memory locations.

Also, certain instructions can cause the contents of the accumulator to be shifted to the right or left. This capability serves many useful purposes. It is one way in which multiplication or division can be implemented.

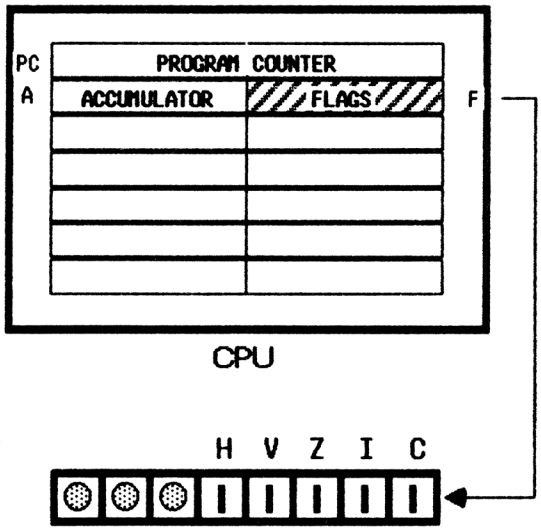
You will deal with the accumulator frequently when doing machine language programming. There will be much to learn about its versatility.

Flags

Flags are single logic cells that can assume one of two states: set or reset (the latter sometimes being referred to as "cleared"). The PC-1500 has five separate flags. While each flag operates independently, as far as the programmer is concerned they can be viewed as being grouped in the five least significant bit positions of an 8-bit *flag register.* This concept is important because at various times it will be important to be able to save or restore the status of all five flags at one time.

Special instructions enable the flag register to be manipulated as an entity to accomplish this objective.

While the flags may be viewed as residing within one register, each flag serves a specific purpose and is operated independently of the other flags. These flags are used to indicate the results of various operations following the execution of certain classes of instructions. It will eventually become necessary for a prospective machine language programmer to thoroughly understand the purposes of each flag. However, for now a brief introduction to their functions will suffice.



FLAGS REGISTER

The *zero flag*(Z flag) is used to indicate if the contents of a register are zero (flag set) or if the contents of a register are non-zero (flag cleared). Note the seemingly inverse relationship here!

The *carry flag* (C flag) is set when there is a carry from the most significant bit of a register and cleared if there is no carry during an addition operation. During subtraction it is set if there is no borrow and cleared if there is a borrow. The carry flag can also be considered as an extension of a register during certain types of operations such as shifts and rotates. It can also be independently set or cleared by special instructions so that its status can be defined at times determined by the programmer.

The *half carry flag* (H flag) is set when there is a carry from the least significant four bits of a register during certain types of decimal arithmetic operations. It is cleared when a carry does not occur from this lower half of a byte. (The eight binary cells that make up a register are sometimes referred to as a *byte*. Did you know that half of a byte is sometimes referred to as a nibble? No joke!)

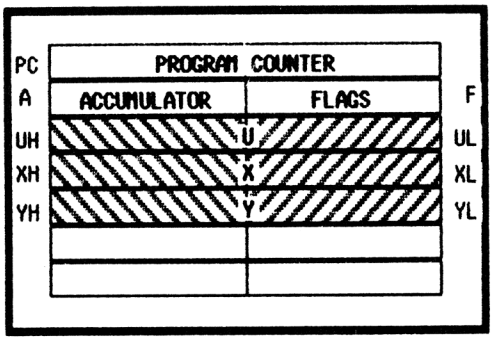
The *overflow flag*(V flag) is set or reset as a function of the carries from the most significant two bits of a register. It has value in certain mathematical procedures. Many ML programmers never have occassion to make use of this particular flag.

The *interrupt enable flag* (I flag) is used to enable or disable a type of CPU interrupt capability provided for the LH5801 CPU. Most programmers will not have to be concerned with its applications.

Data Pointers and General Purpose Registers

The LH5801 has a whole group of registers that may be used for several purposes. While not having quite the versatility of the accumulator (in that they generally lack "calculating" capabilities), never-the-less they come in very handy.

There are six of these *general purpose* 8-bit registers. However, they can be paired to form three sets of 16-bit *data pointer* registers. When serving as a data pointer the contents of the register may be used to indicate an address in memory where data is to be obtained or deposited. Note that when coupled together to form a 16-bit register they can hold any address in the range 0 - 65,535, thus they can point to any valid address in memory that the CPU is theoretically capable of accessing.



When not being used as data pointers, these registers may be used to temporarily store and manipulate information in 8 or 16-bit format. These registers are rather arbitrarily referred to as the X, Y and U register pairs when referenced as 16-bit data pointers. Since each 16-bit pair can also be manipulated as two independent 8-bit groups, each may be referred to as consisting of a high byte and a low byte. Thus the designations XH & XL, YH & YL and UH & UL when the 8-bit registers are referenced as separate entities.

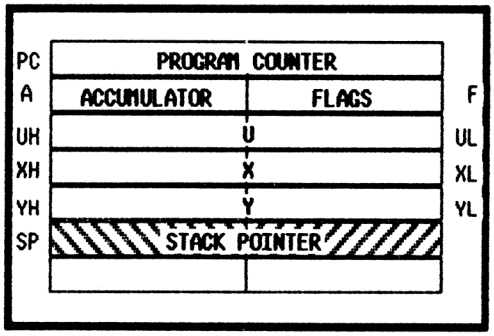
It is worth noting that while a data pointer may be the same 16-bit size as the program counter, their functions are different. Remember, the program counter always tells the CPU where to obtain the next *instruction* in memory. On the other hand, a data pointer register may be used to tell the CPU where to obtain *data* from memory. The two operations are quite distinct and should

 Copyright 1983 POCKET COMPUTER NEWSLETTER

not be confused.

The Stack Pointer

The *stack pointer* is a special 16-bit register that has a variety of interesting uses. Basically, what it does is point to a specific address in memory where information is to be stored on a temporary basis. The stack pointer register is designed such that each time it is utilized by an instruction, its contents are incremented or decremented depending on the type of instruction being performed. This operation provides a method whereby information may be *pushed* into a storage area in memory or *popped* out of that region back into designated CPU registers.



CPU

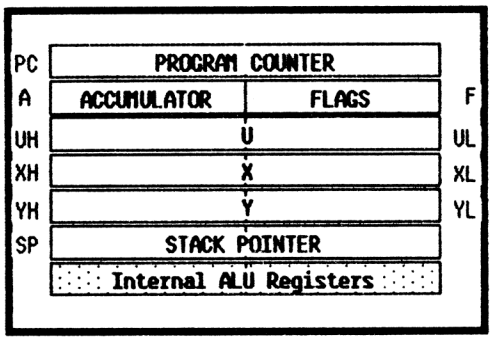
A frequent use of the stack pointer is to save the contents of the program counter (remember what it does?) when a program branches from a main sequence of instructions to a minor sequence. This process is generally known as subroutining. By saving the value of the PC in an area of memory indicated by the stack pointer, the CPU is able to eventually resume operations from the point where the subroutine was first called.

While all of this may seem complicated at this time, don't worry about the details, just grasp the concept. The details of stack pointer operation will be presented when it is appropriate to understand it in depth.

Internal (User Transparent) CPU Registers

The registers that have been discussed are all in some manner or another accessible to the programmer. That is, by giving the proper instructions or sequences of such directives, the contents of those locations can be directly controlled.

There are a few internal registers in the CPU that do a lot of work that is effectively hidden from the view of the user. For instance, there are registers that perform certain types of arithmetic operations. As a group these comprise and are generally referred to as the arithmetic logic unit (ALU for short). There are also registers that control the overall operation of the CPU. One of the most important of these is known as the IR or *instruction register.* This register holds the binary pattern representing an instruction (fetched from memory) while surrounding circuitry "decodes" the instruction. This results in the CPU performing the operation dictated.



CPU

It is often worthwhile remembering that the initial description of a computer as consisting of a CPU and memory is not inaccurate. All a computer really does is *fetch* an instruction from memory and then *execute* the directive it receives. The beauty and power of the computer comes from the fact that it can perform a typical fetch and execute cycle at the machine language level in just a few millionths of a second. That is the case even for a tiny hand-held package such as the PC-1500!

An Instruction at a Time

Just as a computer executes one instruction at a time, a potential machine language programmer should plan on learning about one instruction at a time. While the PC-1500 can execute several hundred different instructions, don't be faint-hearted. Many of the instructions are identical except for the fact that they operate on a different register. Thus, once the concepts for a fundamental class have been learned, you will know how to use a whole group or series of particular directives.

A Treatise on Mnemonics

Mnemonics (pronounced "knee-mon-ics") are memory aids. That is they are a shorthand way of writing machine language directives. They are generally developed in such a way that they help the programmer remember what each instruction does.

The novice programmer is sometimes confused about the true significance of mnemonics. It is important to realize that to a computer at the operational level, there is no such thing as a mnemonic. All the CPU ever sees as it processes instructions are the binary patterns that represent each particular directive. Remember, the CPU has an instruction "decoder" network (circuit) that

translates each different type of pattern into a sequence of events to accomplish a particular objective. Unfortunately, people do not appear to have such efficient decoder networks in their heads *when it comes to manipulating binary patterns.* However, the human mind is quite adept at handling alphanumeric patterns (probably because that is what they are trained or "programmed" to recognize). People have to program computers. Hence, we need to have an efficient method of remembering what kinds of instructions our computers can perform and of writing those directives down so that *we* can easily tell what it is we are doing.

Mnemonics are completely artificial, abstract and arbitrary definitions of machine language instructions. Anybody that wants to can create and compile their own set of mnemonics for any computer. The only reason for having any standardization when it comes to using mnemonics is so that other people with whom we converse and communicate can readily understand what it is we are talking about.

Now it happens that most CPU manufacturers get the honor of promulgating a set of mnemonics to properly represent the instruction set of their beloved CPU. After all, they want people to use their machines. To make life easier on prospective customers, they generally are happy to put forth an easy way to remember all the various machine language instructions that their CPU can perform.

Of course, there are always exceptions, aren't there? When the Sharp PC-1500 was first introduced, it seems that the manufacturer *did not want consumers to know how to program it in machine language!* They thought BASIC programming would be good enough for everybody.

Alas, they were incorrect in their reasoning. A large group of purchasers were indeed interested in utilizing the little PC at its most powerful level. Some of these users, in particular a group of *POCKET COMPUTER NEWSLETTER* readers, worked together to decipher the machine language instruction set that the PC was apparently capable of performing. This was done entirely by analyzing the binary patterns stored in the ROM (read-only memory) portions of the PC.

The principal investigator of this group, *Norlin Rober,* created a set of original mnemonics to represent the instructions that the group discovered. The mnemonics he created are the ones used in this text. They are referred to herein as the *Rober Mnemonics.*

Here is some more information that may help you to further understand the function of mnemonics. If, after you have designed a machine language program using mnemonics, you plan on having your PC execute your directives, you will need to perform a translation from the mnemonics *you* use to the binary patterns (machine code) utilized by the CPU. To do this (for small programs) you will simply use a lookup table. That table will list the mnemonics you use and the equivalent binary codes that must be fed to the CPU to represent each instruction.

Now, if you plan on going into the "big leagues," (such as making your living programming in machine language) then you might eventually want to use a special program that would help you convert from mnemonic format to machine code. Such a program is called an *assembler.* It is nothing more than a program that performs automatic table lookup procedures! Trouble is, it in itself can take up a lot of memory and on a small computer such as a PC can be more of a pain to use than not (due to all the shuffeling of programs in and out of memory, etc.). Such being the case, next thing you know you are talking about using a "cross-assembler." That is an assembler program that works on a larger computer and produces code for the "target" (your PC-1500) machine. Before getting carried away with a discussion about such programs, let me end it by saying that the use of such programs will *not* be assumed in this text. Instead, I shall proceed on the basis that small programs, of the type beginning machine language programmers will feel confident to tackle, will be converted from mnemonics to machine code by the manual (table lookup) method!

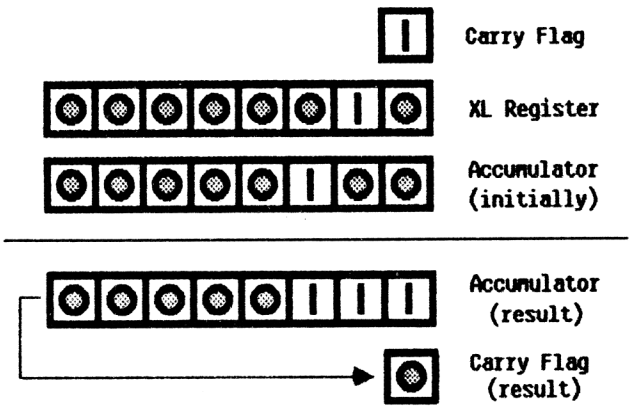
Let's Add Some of This Up

A good way to get a feel for programming at the machine code level is to take an in-depth look at an instruction. We are going to start with a register-to-register addition directive. The machine code representation for this instruction is the 8-bit binary pattern 0 0 0 0 0 0 1 0 which can be represented in hexadecimal numeric shorthand as the value 02. The mnemonic for this command is: ADDA XL. This represents a shorthand method of saying: add the contents of register A (the accumulator) and the contents of the 8-bit register XL (the lower half of the 16-bit register X).

There are some other things you need to know about this directive: (1) The type of addition performed is based on two's complement binary arithmetic, (2) The initial contents of the carry flag (C) are added to the least significant bit position at the start of the addition process, (3) The result of the addition process is left in the accumulator (A register), (4) Any "overflow" from the addition process will be reflected in the status of the carry flag (C) at the conclusion of the operation, (5) The status of the Z, V and H flags (along with the previously mentioned C flag) may be altered by the results of the operation, and (6) The original

 Copyright 1983 POCKET COMPUTER NEWSLETTER

contents of register XL are not altered by the operation.



Gee! That is quite a list of information to have to know just to effectively utilize a single machine language instruction. All of it is important, too. If you forget just one aspect while creating a program, you can end up with a procedure that does not yield correct results. I can tell you from years of experience that one of the items many novice programmers forget about when attempting to apply this type of directive is the affect caused by the initial condition of the carry flag! Note that if the carry flag is set (at a logic one state) when this instruction is executed, that a count of one will be added to whatever is the result of the addition. This feature has considerable practical application. It provides a means of performing what is commonly known in the field as multiple-precision arithmetic. That is a method whereby arithmetic values may be represented by the combined contents of several registers. On the other hand, it can be a subtle (and vicious) trap for the programmer who forgets its ramifications when performing simple register-to-register (often referred to as single-precision) arithmetic. If the status of the carry flag is not known (i.e., reset to the zero state) prior to executing the directive, one can end up with an addition that seems to be mysteriously off by a count of one extra. The reason this operational aspect can really throw beginning programmers is because in a complex program, the status of the carry flag may be varying each time such an instruction is encountered. The net result is a program that capriciously yields incorrect mathematical results!

All of this points out one critical factor about machine language programming. It can be extremely complex and one subtle mistake on the part of a programmer can lead to errors, the cause of which are often difficult to detect. To get a better appreciation of the potential for catastrophe, contemplate the operation of a machine language program with, say, about 10,000 instructions. An error in the application of any one of those could cause incorrect operation of the system. Take it seriously, for indeed that is approximately the number of machine language directives in your PC-1500's BASIC ROM!

On the other hand, note that everything there is to know about the operation of this instruction can be learned and appropriately controlled. Thus, if every aspect is carefully considered and applied, then there is virtually nothing that can go wrong (unless the machine itself should fail, which, short of catastrophic device failure, is an extremely rare occurrence). Execution of the ADDA XL instruction will invariably produce the same result as long as all the parameters remain the same each time the operation is performed.

One of the great marvels of the computer is that once a process has been properly modeled (programmed), no matter how many months or years of development behind it, its future use is timeless. It may have taken five people-years of work to create the coding contained in the BASIC ROM in the PC-1500. However, that coding can now be duplicated in literally millions of devices and used by humankind for eternity (if the manufacturer obliges).

Let us not stray too far from our course. The subject at hand is learning the detailed aspects of invoking the ADDA XL directive.

Another point worthy of note: Take heed of the fact that the carry flag acts as an addition to the least significant bit of the registers at the start of the operation. However, at the end of the procedure it is acting as the overflow handler from the most significant bit position of the accumulator! Thus, in this instance, the carry flag appears to serve opposite ends of a register depending on when one "looks" at the situation. At the beginning of the operation, it is serving as an addend at the rightmost bit position. At the conclusion, it is acting as an extension at the left end of the accumulator -- effectively serving as a ninth bit position.

Finally, it will be mentioned that the ADDA XL instruction takes but one byte of memory to fully specify the directive. Other types of instructions, even variations of the "add to the accumulator" instruction such as this one, can take several bytes of memory as will be outlined shortly.

Learn One, Learn a Bunch

A nice thing about studying the machine codes for a particular CPU is that once you have learned the basic operation of one instruction, you have often learned the key facets of a whole group of instructions. That is certainly the case with the ADDA XL instruction. You see there is a whole group of "add to the accumulator with carry"

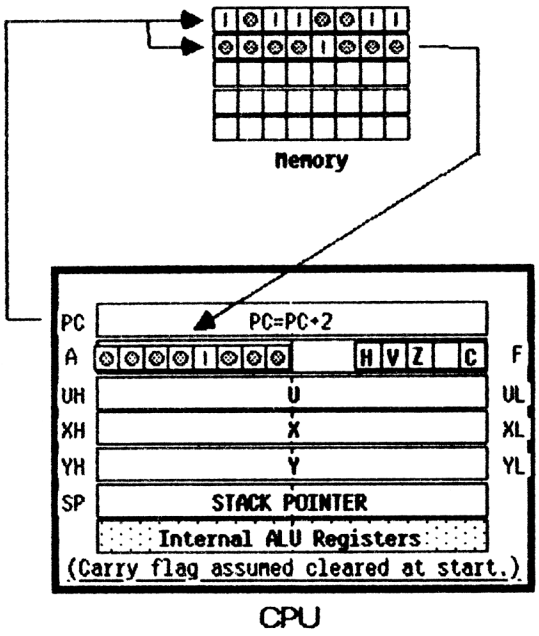
commands:

Mnemonic	Code
ADDA #nn	B3
ADDA (U)	23
ADDA (X)	03
ADDA (Y)	13
ADDA nnnn	A3
ADDA UH	A2
ADDA UL	22
ADDA XH	82
ADDA XL	02
ADDA YH	92
ADDA YL	12

Right there you have 11 different types of ADDA instructions. Can you figure out what each mnemonic represents?

Well, some conventions are being followed with the mnemonics. First of all, the register that follows the ADDA part of the mnemonic is always being added to the A register with the result being left in the accumulator. The register being worked with is left unchanged by the addition operation.

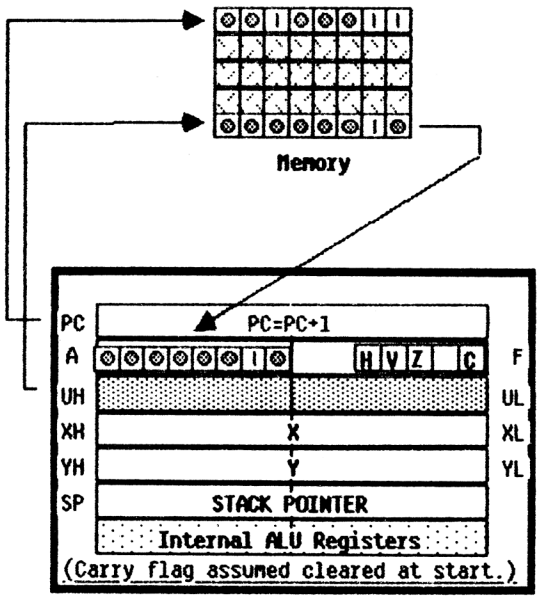
The symbol "#" associated with a mnemonic is a notation that signifies an "immediate" piece of information. An *immediate* piece of data is one that immediately follows the opcode byte. Thus the mnemonic ADDA #nn indicates that whatever value is in the byte that immediately follows the opcode (B3 in this case) will be added to the accumulator. If the hexadecimal value 08 followed the opcode, then 08 would be added to the current contents of the accumulator. Note that this particular instruction thus requires two bytes of memory in order to be fully specified. The code B3, indicating the type of instruction to be performed, must be immediately followed by a second byte



CPU

containing a value that will be used as an operand. In this case the operand will be added to the contents of the accumulator.

An ADDA mnemonic followed by a 16-bit register designation enclosed in parenthesis, such as ADDA (U) means that the specified register is to be used as a *data pointer*. That means that the address contained in the 16-bit register will be the address in memory from which the operand will be taken. In other words, register U (in this case) *points* to the memory address where the actual data byte is located. Note that there is an ADDA instruction for use with each of the three (U, X and Y) 16-bit data pointer registers. Also note that the instruction itself only requires one byte of memory. The operand that is pointed to by the appropriate data pointer register can be any location in memory. In many applications a separate area will be reserved or established in memory to serve as a data storage area. The data pointer will then be set up (prior to execution of the ADDA command) to identify the value that is to be used in the addition operation.



CPU

Can you deduce the format of the directive signified by the mnemonic ADDA nnnn? Sure! It means that the opcode is followed by a 4-digit (hexadecimal) address. The contents of that memory address will contain the value that is to be added to the accumulator. Note that now the instruction uses three bytes of memory in order to be completely specified. One byte for the opcode and two to identify the memory address that contains the data byte.

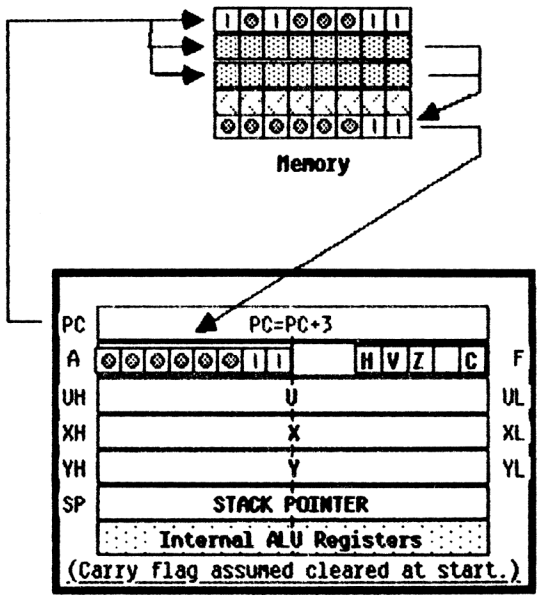
Perhaps this is a good place to delve into thoroughly understanding how many bytes of memory a particular type of instruction is said to

 Copyright 1983 POCKET COMPUTER NEWSLETTER

require. The key here is to realize that the program counter's job is to identify where the next opcode to be executed is located in memory. If an instruction such as ADDA (U) is executed, then the program counter (PC) is merely incremented once to point to the next consecutive memory location. This is because the ADDA (U) directive only uses one byte of memory in which the actual opcode itself is stored.

However, when an instruction such as ADDA #nn is processed, the PC must be incremented twice. Once to advance over the opcode and once to go beyond the immediate byte of data that must be provided after the opcode!

And, the instruction ADDA nnnn will cause the PC to be incremented three times. Once for the opcode, twice for the two bytes following it that hold the 4-digit hexadecimal address where the data value is located.



CPU

In summary, the program counter must automatically be advanced so that it identifies the start of the next instruction that is to be executed. The number of bytes that it must be advanced over for each type of instruction is what we are referring to when we say a directive is a one-, two- or three-byte directive. (While most instructions executed by the LH5801 only require 1 to 3 bytes, there are a few that need even more. Those will be introduced in due course.)

Finally there is a group of ADDA instructions of the type with which you are thoroughly familiar: the group that adds the contents of an 8-bit CPU register (UH, UL, XH, XL, YH or YL) to the accumulator. Note that this latter group consists of data operations that are "intra-CPU." That is, the addition operation takes place amongst registers within the CPU itself. The other types of ADDA instructions discussed all took a data byte from a location in memory.

An Ordered Structure

While this text will not dwell on "hardware" aspects of CPU operation, from time-to-time I may point out features that are related to the internal circuit design. This will be done for the purpose of providing interesting and useful material that may be of value to some machine language programmers.

For instance, let's spend a few moments examining the machine codes that are used to represent the ADDA commands. The codes are shown in hexadecimal notation. The two hexadecimal digits are easily expanded into two adjacent groups of four binary bits. Thus, the hexadecimal code for the ADDA #nn instruction, B3, stands for the binary pattern: 1 0 1 1 0 0 1 1. It is interesting to note that all the ADDA directives associated with intra-CPU register additions end with the hexadecimal digit 2 while all of those that involve data bytes stored in memory end with the digit 3. You can also observe that the most significant hexadecimal digit similarly exhibits interesting structures. For instance, the most significant digit for ADDA (X) is 0, while for ADDA (Y) it is 1 and for ADDA (U) it is 2. Look, too, at how the most significant digit has a distinct pattern amongst the ADDA XH and XL, YH and YL and UH and UL codes. When expanded to a binary representation it is clear that the most significant bit in the opcode is being used to select the high order register of each register-pair.

This information might seem like trivia to beginning programmers, but it can have practical application as you gain coding experience. If you have ever had occassion to watch a seasoned machine language programmer at work (one with several years of concentrated practice) you may have been amazed to notice that the person apparently knew how to directly translate machine code from, say, a raw hexadecimally-coded memory dump. One important way an experienced programmer develops this skill is by learning the significance of particular bit-patterns and the use of individual bits within those patterns.

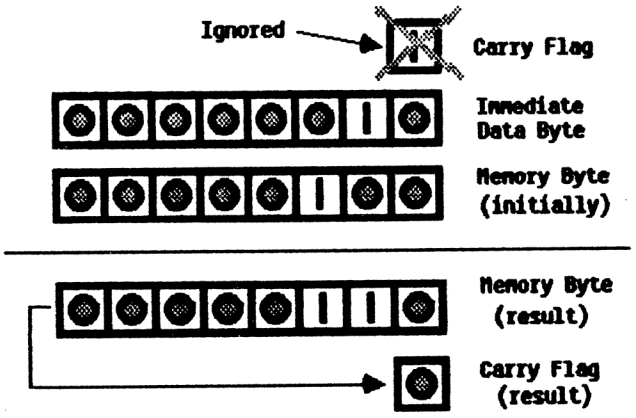
For instance, there are only two other types of PC-1500 ML instructions that end with the hexadecimal digit 2. One of these other types always has the most significant digit in the hexadecimal range C through F. The other uses 4, 5 or 6 as the most significant digit. Knowing this (by heart!) you can instantly classify any machine code ending with a 2! Would you be surprised to know that the group that has 4, 5 and 6 in the most significant digit positions involves the X, Y and U registers in that order? Not if you noticed that the

ADDA group had the most significant digit advancing from 0 to 2 in the same order for registers X, Y and U!

I am not going to recommend that a beginner proceed to put much effort into memorizing this type of information. I am, however, pointing out how it can be done so that you will be aware that the CPU is a logical device, it was designed by a human designer who developed logical patterns to control its operation, and there is nothing magic about the machine. As you gain familiarity with a particular CPU you can start to casually look for such recurring patterns and use the information for what it is worth. If you become a professional machine language programmer, it can be worth a lot of time. However, it is not a necessary skill. You can always look up the meaning of any machine code in a suitable machine code table or better yet, use a computer program known as a disassembler. More about that later.

Add without the Carry

As you might imagine, there are times when a programmer may want to perform addition operations without having to be concerned with the status of the C flag at the start of the procedure. (In other words, without having to be bothered with seeing that the carry flag has been cleared.) The LH5801 CPU has a group of instructions that provides this capability, but it is not as comprehensive as the previously described ADDA category. In fact, it is restricted to commands that add an immediate data value to the contents of a location in memory, instead of the accumulator.

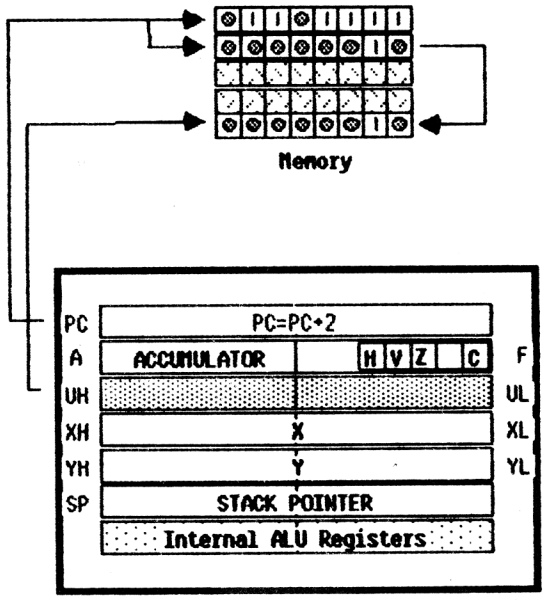


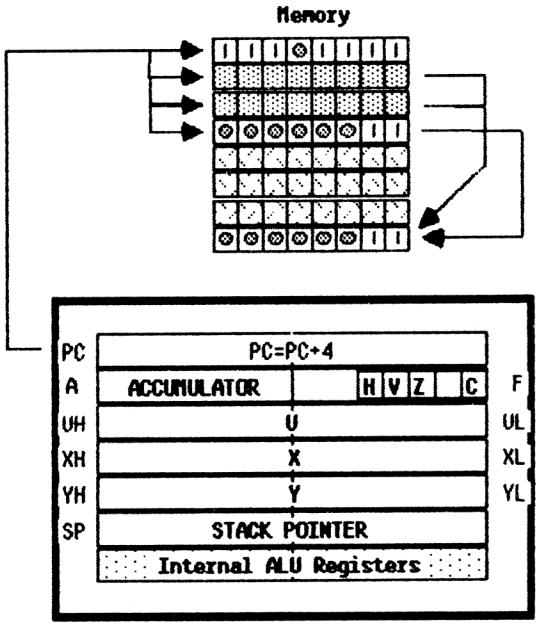
The set has the following mnemonics and machine codes:

Mnemonic	Code
ADNC (U) #nn	6F
ADNC (X) #nn	4F
ADNC (Y) #nn	5F
ADNC nnnn #nn	EF

There are a number of points you must know in order to properly utilize these directives: (1) The addition is performed using two's complement notation, (2) The initial contents of the carry flag are *ignored* at the beginning of the addition process, (3) The result of the addition operation is left in the memory location referenced by the instruction, (4) The status of the carry flag will be determined by the result of the addition operation. It is set if there is an overflow, cleared otherwise. (5) The status of the Z, V and H flags may also be affected by the results of the procedure.

The first three instructions in this group are two-byte directives. One byte representing the opcode, the second the value that is to be added to the specified location in memory. Note that the 16-bit registers U, X or Y must contain the address of the byte in memory to which the immediate data is to be added.





CPU

 Copyright 1983 POCKET COMPUTER NEWSLETTER

The last directive, ADNC nnnn #nn is an example of a directive that will cause the program counter to be advanced by a count of *four* when it is executed. Once for the opcode, twice for the 16-bit memory address in the next two bytes, and a fourth time for the immediate data byte!

Note the machine code pattern. The lower nibble is F for the entire class of instructions. The higher nibble goes from 4 to 5 to 6 to signify use of the X, Y and U registers respectively.

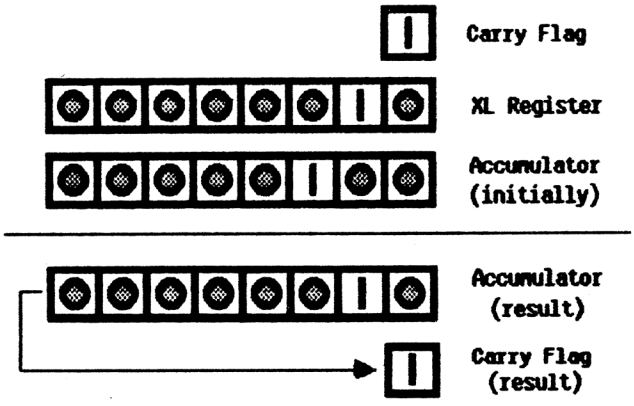
This type of instruction can be particularly valuable when it is desired to advance an address pointer that is being stored in memory. This is especially true when one is in a situation where it might be beneficial not to have to alter the contents of the accumulator.

Subtraction

You will undoubtably not be surprised to learn that the LH5801 CPU is able to perform subtraction. In fact, it has an entire group of directives similar in structure to that of the "add with carry" group that has already been discussed. Here are the mnemonics and machine codes used to represent these subtraction directives:

Mnemonic	Code
SUBA #nn	B1
SUBA (U)	21
SUBA (X)	01
SUBA (Y)	11
SUBA nnnn	A1
SUBA UH	A0
SUBA UL	20
SUBA XH	80
SUBA XL	00
SUBA YH	90
SUBA YL	10

All of the subtraction directives take place between the accumulator and a specified register or memory location. Here are the important things to know about these directives: (1) The arithmetic convention utilized is known as two's complement notation. (2) The complement of the carry flag is considered as a *borrow*. (3) The contents of the specified register or memory location are always subtracted from the contents of the accumulator. (4) The result is left in the accumulator. (5) The original register or memory location value is left unchanged. (6) Any "underflow" caused by the subtraction process will be reflected in the status of the carry flag at the end of the operation. (7) The status of the Z, V and H flags may be altered by the operation.

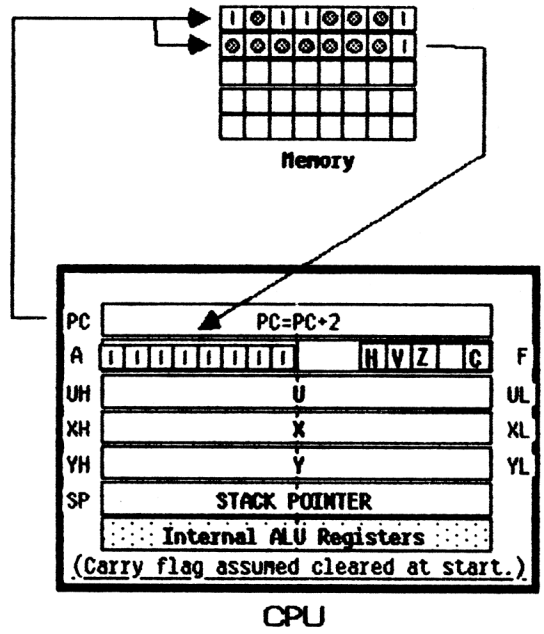


Addressing

As was the case with the addition directives, the subtraction instructions take one, two or three bytes depending on the *addressing* involved. The group of subtraction instructions listed represent four different types of addressing modes.

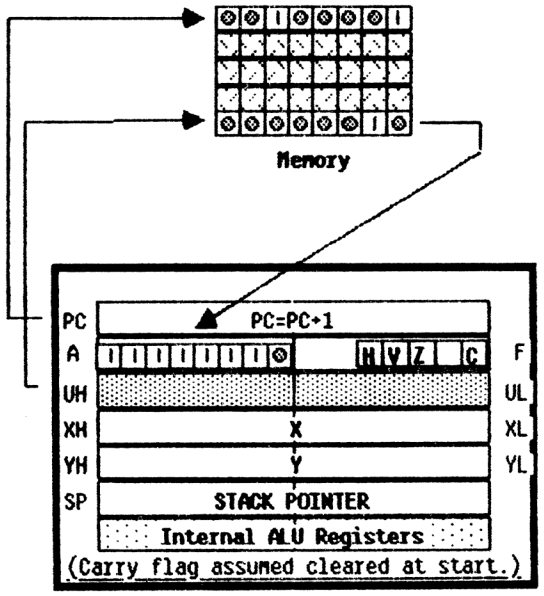
Those such as SUBA UH and SUBA UL utilize what is known as *implied* addressing. That is, the opcode itself specifies the registers or locations that are affected by the operation. The hexadecimal opcode 20 (which is 0 0 1 0 0 0 0 0 in binary) tells the CPU everything it has to know in order to perform the directive. When it detects that pattern, it "knows" that it is to perform a subtraction operation using the accumulator and the 8-bit UL register. In a like fashion, the pattern 1 0 1 0 0 0 0 0 (A0 hexadecimal) tells the CPU to perform the subtraction between the 8-bit UH register and the accumulator. These types of directives only use one byte of memory since the addressing information (related to what registers will be affected) are inherent within the actual machine code. In fact, this mode of addressing is sometimes also referred to as *inherent* or *register* addressing.

The directive SUBA #nn is an example of *immediate* addressing. The byte that immediately



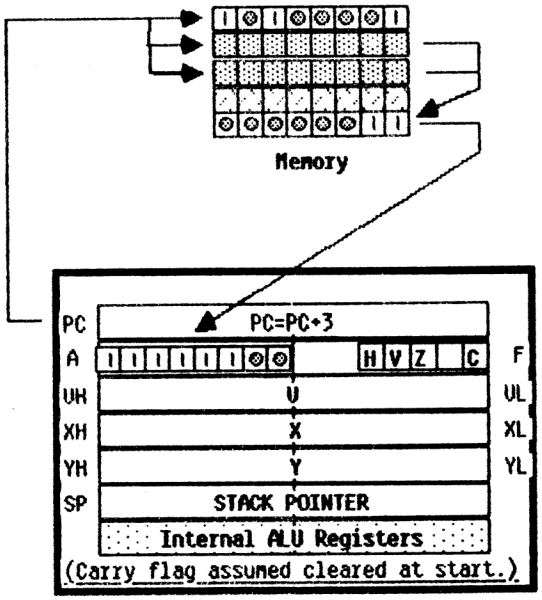
follows the opcode is the location that is to be subtracted from the accumulator. This directive requires two bytes of memory. One for the opcode, the other to hold the data that is to be subtracted.

The instructions SUBA (U), SUBA (X), SUBA (Y) all represent examples of what is often referred to as *register indirect* addressing. This is because the contents of the 16-bit register U, X or Y holds an address of a memory location where the actual data that is to be subtracted is stored. Note that the U, X or Y registers themselves do not hold the data, rather they *point* to the data location.



CPU

Finally, SUBA nnnn is illustrative of the mode known as *direct* addressing. In this mode, the two bytes that follow the opcode contain the actual address indicating where the data byte is stored.



CPU

Note that these two address bytes are considered as part of the instruction. Thus, this directive takes three bytes of memory to be properly defined.

These *addressing modes* (and there will be a few others introduced later) are an important concept. Most types of instructions have several different possible addressing modes. If you go back now and review the ADDA group of instructions you can observe that it has the exact same set of addressing modes.

Some instructions utilize a combination of addressing modes. The previously discussed ADNC (U) #nn utilizes a *register indirect - immediate* mode. The 16-bit register (U in this case) points to a memory location and the immediate data byte holds the data that is to be added to the contents of the referenced memory location. Can you figure out what combination of modes are used in the ADNC nnnn #nn instruction? Right! It combines *direct* and *immediate* modes.

Can you find any connection between the machine codes used (in the opcode byte) and the addressing mode being invoked? Remember, the CPU was created by a logic-oriented circuit designer. Do you think that perhaps a particular bit position within the opcode byte is frequently used to signify a particular addressing mode? You might keep an eye out for address-mode patterns (in the opcodes) as additional types of instructions are introduced. This is another piece of information that the "pro's" look for as they become intimately familiar with the operation of a particular CPU.

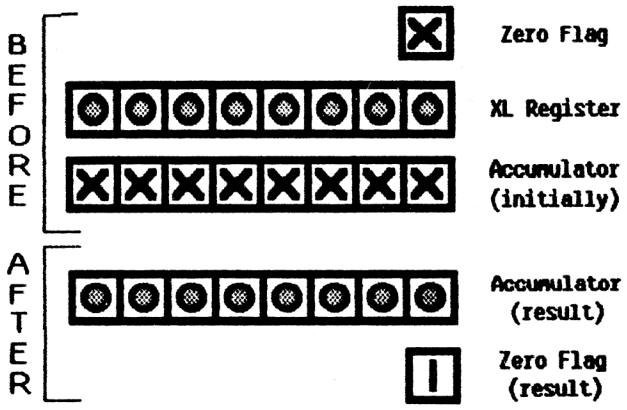
Loading the Accumulator

Adding and subtracting quantities in a computer is all well and good, but you need to have values to work with in the first place. How do you obtain values and get them positioned in the proper registers? You use *load* and *store* directives. These are instructions that cause information to be moved about from one location to another. They are probably amongst the most often used commands. Right now we will become familiar with the group of directives that may be used to place information into the accumulator or A register:

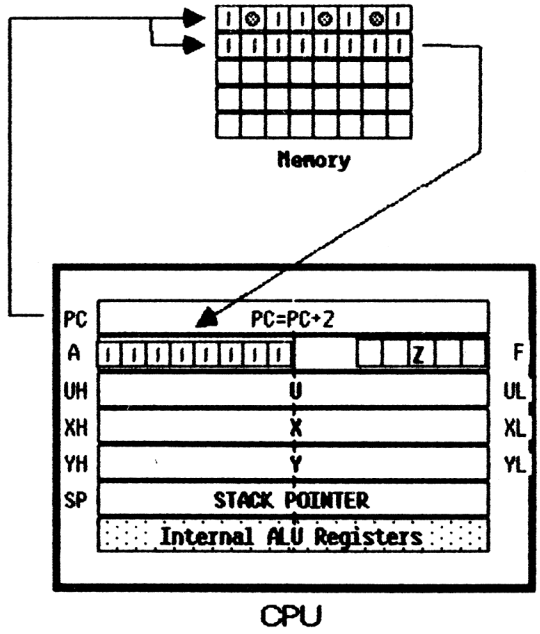
Mnemonic	Code
LDA #nn	B5
LDA (U)	25
LDA (X)	05
LDA (Y)	15
LDA nnnn	A5
LDA UH	A4
LDA UL	24
LDA XH	84
LDA XL	04
LDA YH	94
LDA YL	14

 Copyright 1983 POCKET COMPUTER NEWSLETTER

Here is what you need to know about this group of instructions: (1) A "copy" of the information in the specified memory location or register is always transferred (loaded) into the accumulator. (2) The status of the Z (zero) flag will reflect whether or not the value transferred was equal to zero. (3) The other flags are not affected by this type of operation.



There, that was easy wasn't it? All these instructions do is provide a means of moving information about in the computer system. Note that you can transfer data from the CPU registers and (using several addressing modes) from locations in memory. This particular group always places the data into the accumulator.



You might also want to take note of the fact that only one CPU flag is affected by this type of directive: the zero (Z) flag.

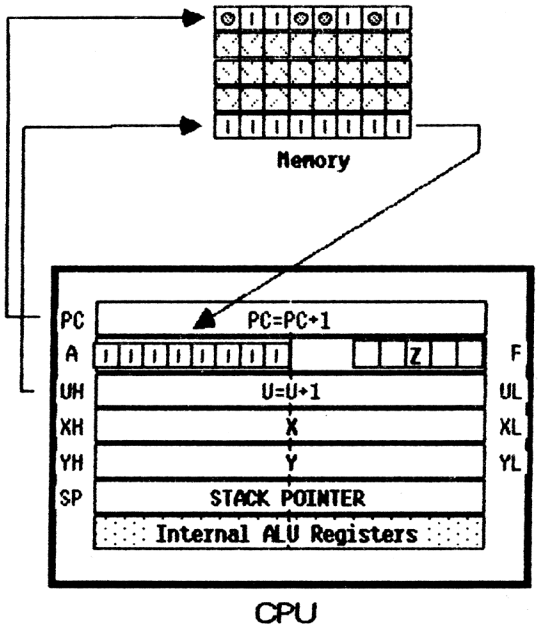
Can you think of any reasons why the designers allowed only this flag to be affected by the load operations?

There are several good reasons. One is that some of the other flags, such as the carry, need to be maintained when multiple-precision arithmetic is being performed. (That is arithmetic where the numbers being manipulated are stored in two or more bytes, yet are considered as one value.) Because these numbers take more than one byte in which to be stored, they have to be processed in byte-sized sections. If a load directive interfered with the status of, say, the carry flag, then the add with carry instructions would be worthless during such operations.

But why let even the zero flag be influenced by load operations? Because, as will be illustrated later, being able to detect a "zero byte" (a byte in which all the bits are cleared to zero) has many practical applications. It is often used to denote the end of data in lookup tables, to mark the end of a string of stored text, and so forth. So, being able to easily detect the presence of a zero byte when it is first loaded into a location, can have considerable practical programming value. The LH5801 designers knew this and thus permitted the zero flag to be affected by this operation, while protecting all other flags.

The load directives are used so extensively in machine language programming that it is common to provide supplemental types of instructions. For instance, the CPU used in the Sharp PC-1500 has the following special load directives:

Mnemonic	Code
LDAI (U)	65
LDAI (X)	45
LDAI (Y)	55
LDAD (U)	67
LDAD (X)	47
LDAD (Y)	57



These are known as *load and increment* and *load and decrement* directives. They transfer a byte from the memory location pointed to by the contents of the 16-bit registers (U, X or Y) into the accumulator *and then they automatically increment or decrement the pointer register by a count of one!* These powerful instructions are most useful when it is desirable to sequentially process a whole block of memory.

Only the Z (zero) flag can be affected by these directives, in exactly the same manner as for the LDA (U) instruction. That is, it is cleared if the byte transferred is non-zero and set if it is zero. Notice that the increment or decrement of the pointer register does not affect the flags. *It is the byte being transferred* that controls the status of the Z flag when any of this group is executed.

All of the load instructions discussed so far have been designed to transfer data *into* the accumulator. But, how about getting data into the other CPU registers?

Loading Other CPU Registers

Each half of the 16-bit registers U, X and Y can be loaded with immediate data by the following load instructions:

Mnemonic	Code
LDUH #nn	68
LDUL #nn	6A
LDXH #nn	48
LDXL #nn	4A
LDYH #nn	58
LDYL #nn	5A

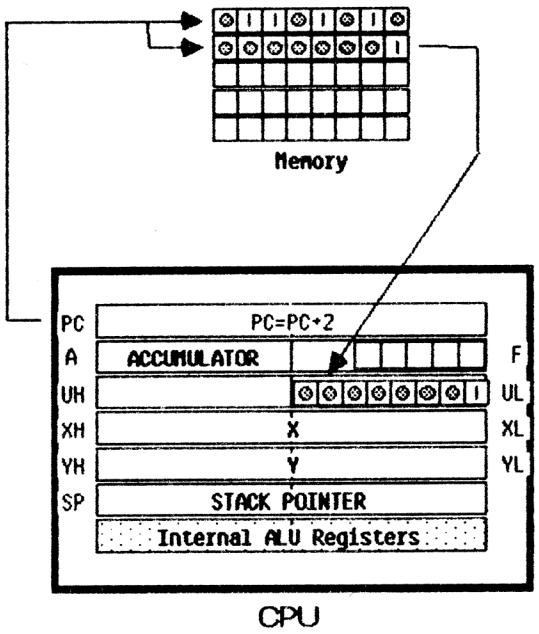
One important difference between this group of directives and the previously described load commands that dealt with the accumulator is that *none of the flags* are altered by these transfers, not even the zero flag.

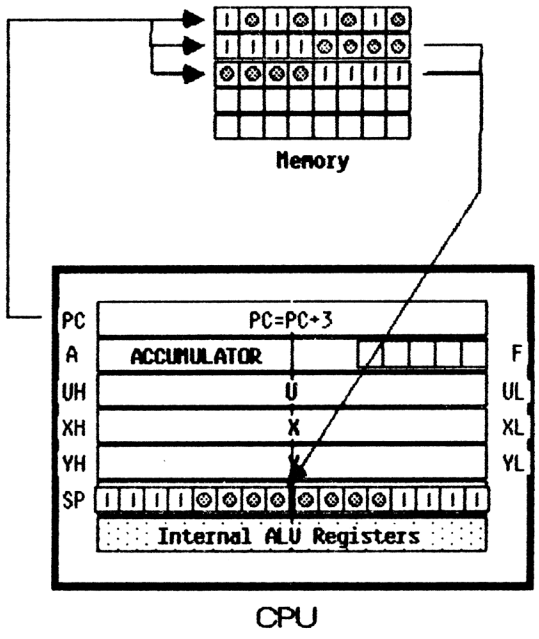
The fact that each half of the 16-bit pointer registers can be loaded with a byte has many practical applications. Frequently these registers are used as temporary storage locations within the CPU. Being able to treat these registers as consisting of two independent bytes (when it is desired by the programmer) means that more data can be readily accessed within the CPU.

On the other hand, you must remember that if you want to set up a 16-bit register so that it contains a memory address, then you must use a sequence of two load immediate instructions. One to load the most significant half (UH, XH or YH) and the other to load the least significant half (UL, XL or YL).

It would have been theoretically possible for the CPU designer to have provided a load immediate instruction that loaded all 16 bits of the data pointer registers at one time. The advantage of opting for the 8-bit loads is flexibility. The trade-off is that it takes two loads to completely define an address. (However, as you will learn later, this is not as serious a trade-off as it might initially appear. In practical programming situations it turns out that the most significant portion of an address does not have to be changed as often as the least significant portion. Thus, many times, the address in a data pointer register can be altered to the desired value just by changing one byte.)

There is, in fact, a 16-bit load immediate directive provided in the instruction set of the CPU being discussed. It is used to load the stack pointer with a complete 16-bit address.





 Copyright 1983 POCKET COMPUTER NEWSLETTER

Mnemonic	Code
LDS# nnnn	AA

This is the *only* directive in the instruction set where the two bytes following the opcode both act as immediate data values. Normally these values represent an address that is loaded directly into the stack pointer. None of the CPU flags are affected by this command. (The stack pointer will be discussed in detail at a more appropriate point. Have patience, please.)

Storing the Accumulator

So far, all the "load" instructions have been transferring information in one direction: into the accumulator or from a location in memory to a register in the CPU. It would be a pretty useless computer if data could only be transferred into the accumulator or CPU and not out. Of course, that is not the case. There is another group of instructions that can transfer information in the other direction. In some systems, this group is still referred to as "load" directives. Here, however, to differentiate between the directions of data flow, an instruction that passes data *from* the accumulator to a location in memory, will be designated a *store* directive.

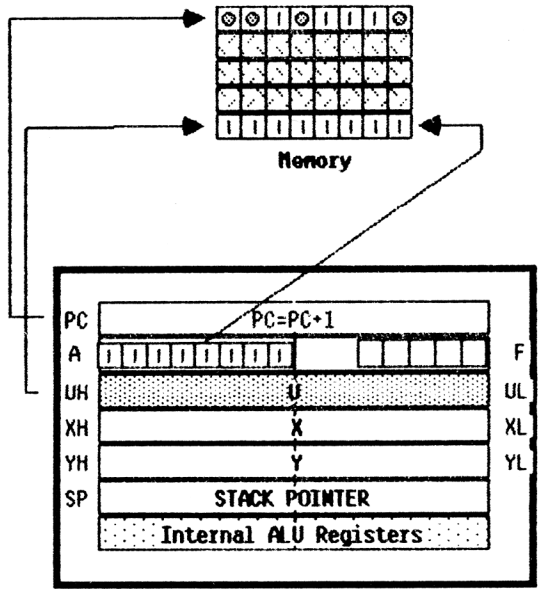
The first of the store directives to be discussed are those that simply transfer the contents of the accumulator to a location in memory or to an 8-bit register within the CPU:

Mnemonic	Code
STA (U)	2E
STA (X)	0E
STA (Y)	1E
STA nnnn	AE
STA UH	28
STA UL	2A
STA XH	08
STA XL	0A
STA YH	18
STA YL	1A

You need to know the following in order to use these commands effectively: (1) A copy of the A register (accumulator) is placed into the indicated memory location or 8-bit CPU register. (2) *None* of the flags are affected by these operations. Not even the zero (Z) flag.

Thus, the CPU can store results from the accumulator into memory or another CPU register without altering the status of any of its flags. This ability to accomplish such transfers without altering the flags is useful in many circumstances. Note that now not even the Z flag is sacrificed. This is because when *constructing* tables or text strings, the CPU can insert a zero byte to serve as a terminator. It does not have to *detect* such a marker as is typically the case when memory is being scanned. The distinction will be made clearer later.

Remember, the STA (U), (X) and (Y) directives use the contents of those 16-bit registers to *point* to the location in memory where the byte will be saved.



CPU

The STA nnnn command uses the 16-bit value (nnnn) as an absolute memory address in which to store the data.

The store directives also include the so-called *auto-increment* and *auto-decrement* groups:

Mnemonic	Code
STAI (U)	61
STAI (X)	41
STAI (Y)	51
STAD (U)	63
STAD (X)	43
STAD (Y)	53

These are similar to the load directives. After a byte is transferred from the accumulator to the memory location pointed to by the address in the U, X or Y register, the contents of the data pointer register is automatically incremented (by one) or it is decremented (by one). It is thus ready to point to the next sequential location in memory. Naturally, this is quite beneficial when it is desired to pack data into a continuous block of memory. However, unlike the load instructions, the CPU flags are not altered by these store commands.

In and Out

Finally, to wrap up the presentation of the load and store instructions, there is a real whiz-bang directive. This command does the equivalent of

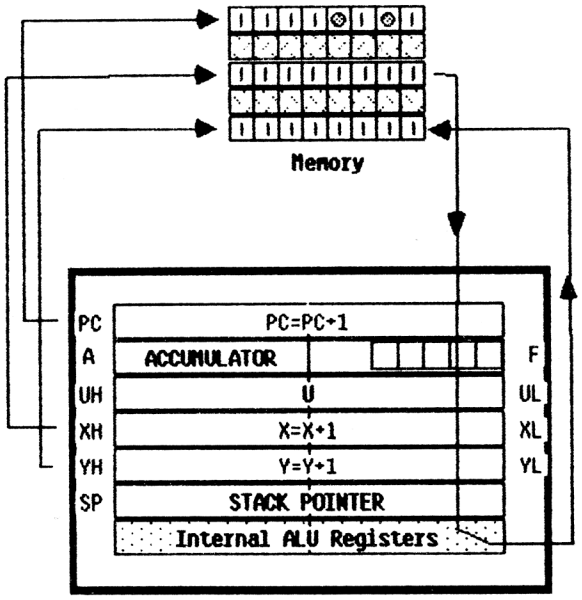
both a load and a store plus it automatically increments *two* data pointer registers! It is a real block-buster -- and that is exactly what it is intended to do -- move large blocks of memory quickly and conveniently.

Mnemonic	Code
STI (X)(Y)	F5

The byte of data stored at the memory location pointed to by the contents of 16-bit CPU register X is first loaded into an *internal* CPU register. It is then stored into the memory location pointed to by the contents of the 16-bit CPU register Y. Next, the contents of both X and Y are advanced by a count of one.

None of the CPU flags are affected by the operation of this instruction.

It is a super instruction to use when you need to move blocks of data from one location in memory to another! This type of directive is indicative of advanced microprocessor designs. Earlier CPU chips did not have such powerful operations combined into a single machine language opcode.



CPU

A Program

Only a small fraction of the commands that are available in the repertoire of the LH5801 have been presented at this point. There is much more to learn. But perhaps now is a good time to get a glimpse at what the future looks like for a machine language programmer. More than enough instructions have been explained so that we can demonstrate a rudimentary, but actual, program. The purpose at this point is to build confidence and provide some feedback for the serious student. Take heart, what you are learning really works. You can make things happen with this information!

The goal at this point is to develop a series of instructions that will accomplish the following: Add one number to another and store the result in a location in memory.

Simple enough, eh? How many ways do you think that could be accomplished using just the machine language directives you have learned so far? There are quite a few, believe me. In fact, that is one of the joys of working at the machine level. There are usually a variety of ways of accomplishing any given task. The "best" way often depends on your objectives. Sometimes, there is really no such thing as a best way. There are, instead, simply a host of ways to accomplish the same end result.

Before reading further, you might just want to take a few minutes to write down some of the ways that you can think of to accomplish the goal: add two simple numbers together and store them in memory.

Did you think of the following method?

(1) Load one value into the A register.

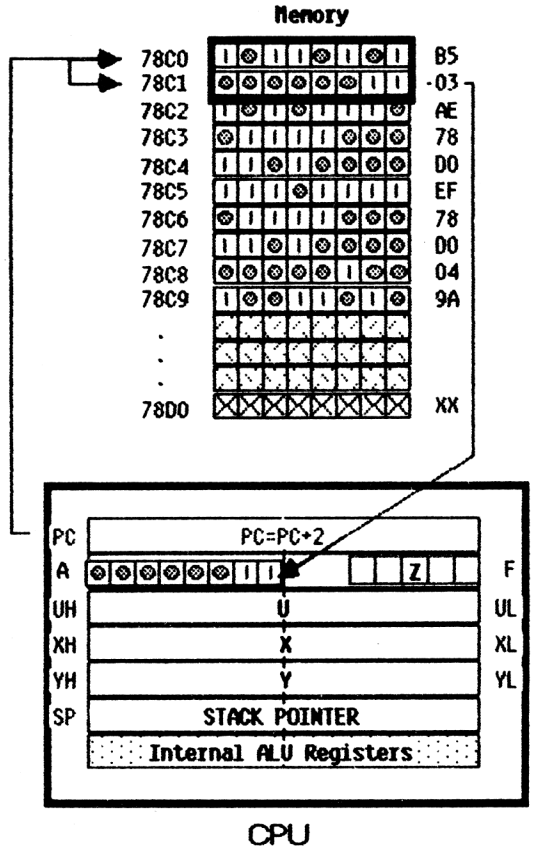
(2) Store the contents of the A register (which now contains the first value) into a specific memory location.

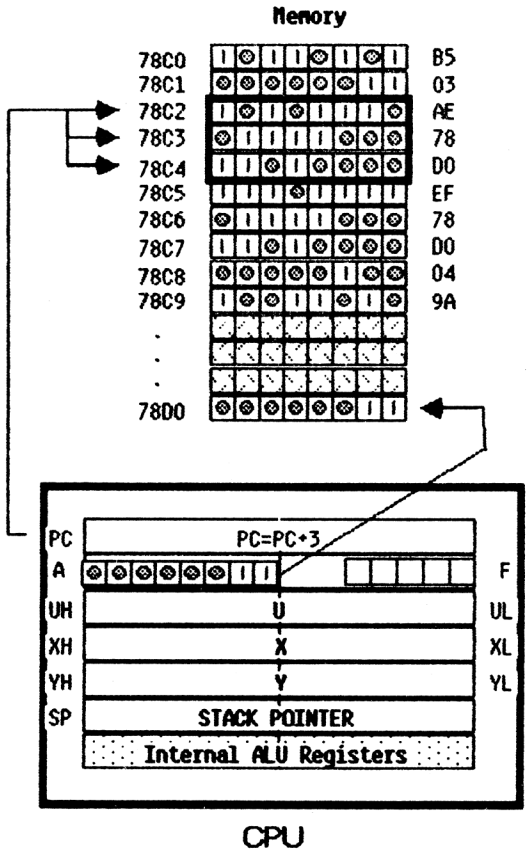
(3) Add the second value to that memory location, leaving the result there.

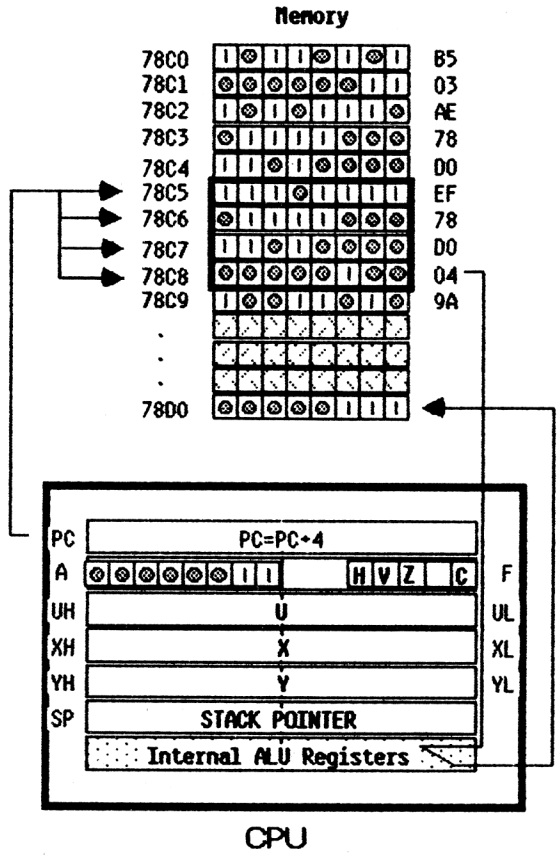
One reason for choosing this method is that it is not affected by the status of the carry flag at the start of the operation. Do you recall that all of the instructions that add something to the *accumulator* are dependent upon the initial status of the carry flag? How do you know, at any given point, what the status of the carry flag is (or will be)? If you don't know what it is, should you be using an instruction that is dependent upon its status? (Actually, there are instructions that can be used to place the carry flag in a known state. And, it is often possible to logically control the state of the flag at a particular point in a program. However, if you don't know what those methods are, then you had better be careful when dealing with "add with carry" commands!) Did you choose a safe way of accomplishing the given task or would you have been on unsure grounds? *You were on unsure grounds if you elected to use any add with carry directive and you did not specifically devise a way of insuring that the carry flag was set to a known state prior to performing the addition!*

The first rule of machine language programming is that you have to *know* what is going on. Guessing or hoping that conditions will be this way or that way will not work. If you can't be sure about the status of something, then you had better design things so that you end up *knowing* the status. If you cannot logically figure out a way of accomplishing this, then the machine cannot do it either. (You see, *you* really *are* the boss.)

Three directives would accomplish the stated goal: LDA #nn, STA nnnn, ADNC nnnn #nn. Load the accumulator with the first number. Store the accumulator in the desired memory location. Add







(no carry) the second number to the memory location where the first number was stored.

If the numbers to be added were 3 and 4 and the hexadecimal address of the storage location in memory was 78D0, here is how the machine code for this series of instructions would appear (in hexadecimal notation): B5 03 AE 78 D0 EF 78 D0 04.

The accompanying pictorials illustrate each step of the program's operation.

You can try this program out for yourself by loading it into a PC-1500 or PC-2. One way to do this is to use BASIC POKE statements. Here is a BASIC program that will load and then execute those machine language directives:

```
10:POKE &78C0,&B5
   ,&03,&AE,&78,&
   D0,&EF,&78,&D0
   ,&04,&9A
20:CALL &78C0
30:PRINT PEEK &78
   D0
40:END
```

Line 10 of this program places the actual machine codes (using hexadecimal notation) into the PC's memory. These instructions are being stored in the portion of memory normally used to hold the BASIC variable A$, starting at the hexadecimal address &78C0. The values following the four-digit address are the actual machine

codes for the program in hexadecimal format. (The last byte in line 10, &9A, is the code for a machine language directive that will cause the machine language program to be exited back to BASIC.)

Line 20 causes the machine language codes to be executed starting at hexadecimal address 78C0 (where the program was stored). After executing those machine code directives, control returns to the BASIC program.

Line 30 causes the PC to display the contents of the memory location having the hexadecimal address 78D0. This is the first byte of the area in which the BASIC variable B$ is normally stored. The machine language program uses this location to store the first number (from the accumulator). It is also the location where the ADNC instruction takes action and leaves the results of the addition (without carry) operation.

Line 40 denotes the end of the BASIC code.

Executing this BASIC program by placing the PC in the RUN mode and executing GOTO 10 or RUN should result in the value 7 being displayed.

Do you understand everything that is going on at this point? Does everything in the descriptive pictorials that portrays the operation of this machine language procedure make sense? If not, now is a good time to review the earlier parts of this text.

By the way, attempting to examine the operation of machine language routines by using PEEK and POKE directives is a very slow and tedious method. A far better tool to use is a machine language monitor program. This is a special program designed to facilitate working at the machine code level. If you plan to make a serious study of this discipline, I strongly recommend that you obtain a copy of the **Loader/Monitor/Disassembler** package sold by the *POCKET COMPUTER NEWSLETTER*. This is a powerful tool that makes it easy to place machine codes into memory, examine the contents of memory, execute machine language routines, and so forth, in a direct mode, without having to translate through PEEK and POKE directives. The disassembler program which is part of this package utilizes the *Rober Mnemonics* which are used in this text. It is able to translate machine codes directly into these mnemonics. Hence, it is highly useful for verifying that machine language routines have been properly loaded into memory as well as for use in exploring "uncharted" areas of memory (such as ROMs). The loader part of this package makes it easy to put together sections of a machine language program. Plus, the package provides a convenient means of saving and restoring machine code programs by using a cassette recorder. No serious MLP student should be without such a tool.

Is learning to program in machine language worth all the trouble? Not for everybody. It depends on what you want to end up being able to do. You now can probably begin to see that dealing at the machine level greatly complicates matters from the programmer's view. It took nine bytes of memory storage just to specify the adding together of two tiny numbers. Even with such a simple task, one had to be concerned with the use of the carry flag. The work is greatly compounded when one has to start dealing with larger values or non-integer quantities.

But, you can probably also gather that the degree of control at this level is fantastic. You are able to dictate every aspect of the machine's operation. Streamlining the flow of operations to accomplish a specific job is just one noticeable benefit. Speed of operation is another. Do you realize that those three instructions used to add together two simple numbers can be performed in about 25 millionths of a second? You could string together some 40,000 similar sequences and still have them all performed in less than a second!

If you still want to hang in there and learn more about the subject, then read on. It is time to describe some of the logical operations that the LH5801 CPU can perform.

The Logical AND

The next few classes of instructions that will be discussed are those known as *Boolean logic* operations. In the LH5801 these mathematical operations are always performed on the contents of the accumulator or on the contents of a location in memory.

The ability to perform these types of logic operations are valuable in many applications. Indeed, they give the computer the ability to duplicate the type of electronic logic found in modern electronic digital circuitry.

These logic operations are always performed on a bit-by-bit basis between the accumulator (or a memory register) and the operand byte. The results of the operation are stored in the accumulator (or memory register). Furthermore, the status of the Z (zero) flag will be affected by the results of the logic operation. Thus, these types of directives, among others, ultimately provide the computer with a means of modifying its own behavior *depending on its findings* as it examines data.

The first group of Boolean logic operations to be presented are those that perform a logical AND operation between the accumulator (A register) and another byte of data in memory.

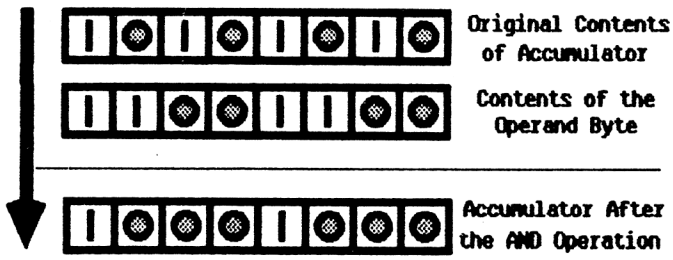
Mnemonic	Code
ANDA #nn	B9
ANDA (U)	29

 Copyright 1983 POCKET COMPUTER NEWSLETTER

ANDA (X)	09
ANDA (Y)	19
ANDA nnnn	A9

Note that there are three types of addressing provided: immediate, register indirect, and direct. To save space in the future, pictorials showing the addressing modes for each class of instruction will not be provided. Refer to the earlier diagrams when necessary to refresh your memory concerning addressing relationships.

The execution of one of the above Boolean AND directives by the LH5801 CPU does the following. Each bit position in the accumulator is compared to the corresponding bit position in the operand byte. As this is done, a logical AND operation is performed between the identically-positioned bits. If both the bit in the accumulator and the bit in the operand are set to the logic 1 state, then the bit in the accumulator will be left in the logic 1 condition. For all other possible combinations (the bits are opposite in state or both are zero), the bit in the accumulator will be left at the logic 0 (cleared) state. If all of the bits in the accumulator end up being zero, then the Z flag will be set. Otherwise it will be cleared. None of the other CPU flags are affected by the operation. The contents of the operand byte are not altered by the procedure.



The logical AND operation may also be performed upon the contents of a particular memory location through the following commands:

Mnemonic	Code
AND (U) #nn	69
AND (X) #nn	49
AND (Y) #nn	59
AND nnnn #nn	E9

The same procedures apply with these directives except that now the operation takes place between the designated memory location and the immediate data byte that is part of the instruction. The results are left in the designated memory site. Only the Z flag is influenced by the operation.

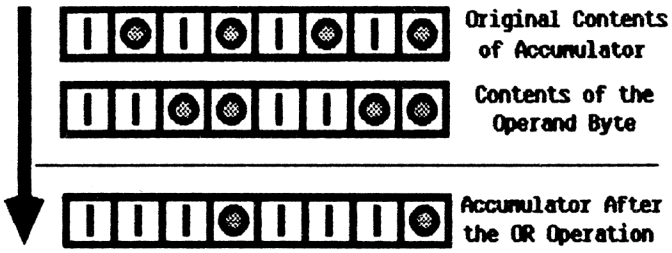
Logical AND operations are particularly useful for performing what are known as *masking* or *stripping* operations. That is, when it is desired to eliminate just a portion of a register's contents. For instance, if one wants to retain just the four least significant bits of an 8-bit register, the entire contents of that register may be ANDed with a register containing the hexadecimal value 0F. Try it yourself (mentally or with the help of pencil and paper) to see how the unwanted half of the byte is reduced to zero. Masking or stripping operations are commonly used to pack data, manipulate binary-coded-decimal (BCD) values and format data inputs/outputs.

The Logical Or

As was the case for the logical AND directives, the logical OR instructions may be divided into two groups: those that leave the result in the accumulator and those that leave the result in a memory location. The first group has the following mnemonics and codes:

Mnemonic	Code
ORA #nn	BB
ORA (U)	2B
ORA (X)	0B
ORA (Y)	1B
ORA nnnn	AB

The Boolean OR operation is also performed on a bit-by-bit basis. For the above instructions, the operation takes place between the accumulator and the designated operand byte with the results being left in the accumulator. A bit position in the accumulator is set to a logic 1 if either it *or* the corresponding bit position in the operand is set to a logic 1. Note that the case where they both are a 1 also satisfies the condition. But, if neither register contains a 1 in a given bit position, then the accumulator bit for that position remains in the logic zero state.



The zero (Z) flag is affected by the results left in the accumulator following the execution of one of these directives.

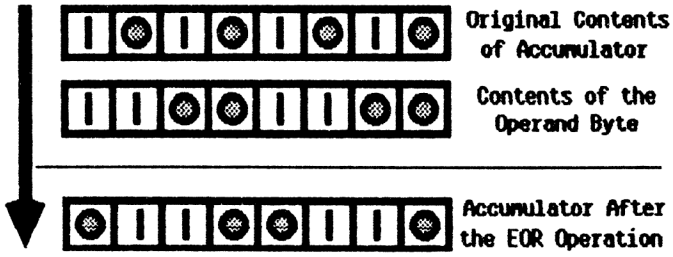
A second group of instructions is performed with and stores the results of the logical OR operation in a designated memory location:

Mnemonic	Code
OR (U) #nn	6B
OR (X) #nn	4B
OR (Y) #nn	5B
OR nnnn #nn	EB

Again, the Z flag is affected by the results of the procedure.

The Logical Exclusive OR

There is a variation of the logical OR operation that the LH5801 is able to perform. It is known as the exclusive OR. It is similar to the OR operation. The difference is that when the corresponding bits between the operand and affected register are both a 1, then the bit position in the results register is cleared to the zero state. Stated another way: a bit in the results register is set to a logic 1 *if, and only if,* just *one* of the registers has a 1 in that bit position.



Because this class of instructions is more limited in application, the CPU designers limited it to operating with the contents of the accumulator and an operand byte. There are no exclusive OR directives that use a memory location as a results register, as there was for the AND and OR classes.

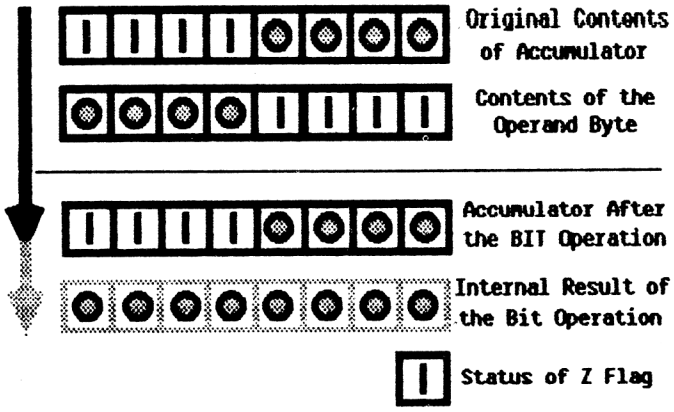
Mnemonic	Code
EORA #nn	BD
EORA (U)	2D
EORA (X)	0D
EORA (Y)	1D
EORA nnnn	AD

The Z flag is the only CPU flag affected by the results of these operations.

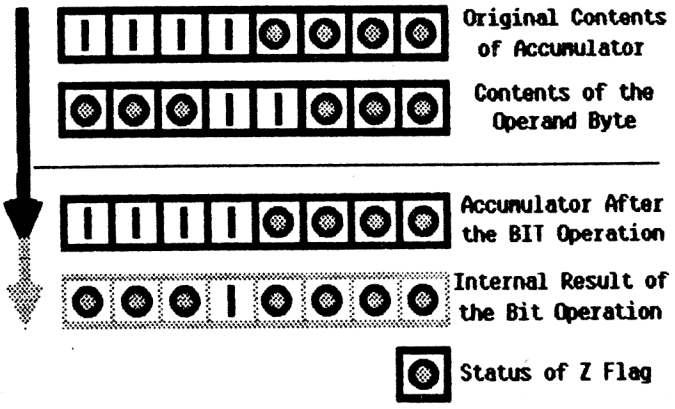
The Logical Bit Test

Another class of instructions provided by the CPU used in the PC-1500 and PC-2 is really just a specialized form of the logical AND operation. The directives in this group enable a programmer to determine which individual bit(s) within a register are set to the logic 1 state. This can be done *without actually altering the contents of the register being examined.*

Known as the *logical bit test,* this type of instruction performs a logical AND between the contents of the accumulator or a location in memory and the contents of another register. The second register can be a location in memory or represented by an immediate data byte, depending on the type of addressing being used. However, the contents of the accumulator or location in memory are not actually altered by the test. Instead, an internal CPU register holds the result of the logical AND operation. If the result held in this internal register is that all the bit positions are zero, then the zero (Z) flag is set.


Original Contents of Accumulator
Contents of the Operand Byte
Accumulator After the BIT Operation
Internal Result of the Bit Operation
Status of Z Flag

If any bit position is in the 1 state, then the zero flag is cleared.


Original Contents of Accumulator
Contents of the Operand Byte
Accumulator After the BIT Operation
Internal Result of the Bit Operation
Status of Z Flag

This type of instruction is particularly useful in determining if a particular bit within a byte is set. Note, however, that it is the status of the Z flag alone that reflects this information. None of the bits within the accumulator or memory are altered. This operation is an example of what is often called a *non-destructive* test or comparison. This is because none of the information involved in the test is altered or lost.

If you compare the operation of this type of directive against that of the regular logical AND, you will note that the value in the results register (the accumulator or memory location) can be altered by the procedure.

Here are the bit test instructions that the CPU is able to execute in a PC-1500 or PC-2:

Mnemonic	Code
BITA #nn	BF
BITA (U)	2F
BITA (X)	0F
BITA (Y)	1F
BITA nnnn	AF

 Copyright 1983 POCKET COMPUTER NEWSLETTER

BIT (U) #nn	6D
BIT (X) #nn	4D
BIT (Y) #nn	5D
BIT nnnn #nn	ED

Compares

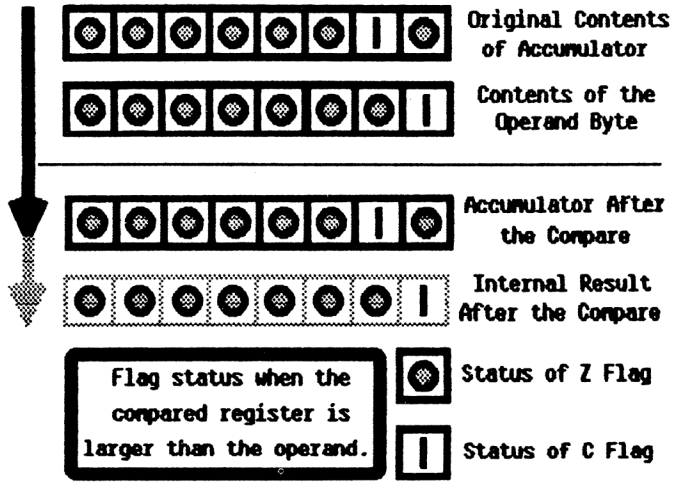
The next series of instructions to be discussed are similar in one concept to the bit test group. That is they also utilize an internal (virtual or invisible) register to store the result of a comparison. The status of the CPU flags are then changed to reflect the conditions in this internal register. None of the regular CPU registers or memory locations that are accessible to the programmer are actually modified by the operation of these instructions.

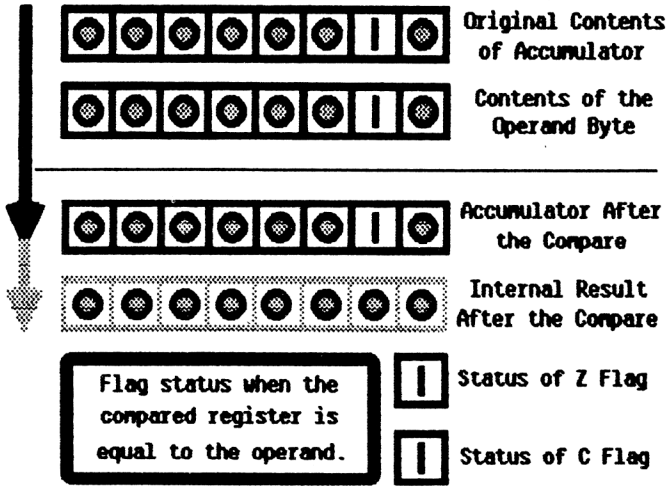
However, these instructions differ in the manner in which the comparison is made. In the bit test group, a logical AND was used to compare bits. In the compare directives, the operand value is *subtracted* (using the two's complement method) from the accumulator, another CPU register or a memory location. The results of this subtraction are held temporarily in an internal (invisible) CPU register. The carry (C) and zero (Z) flags are then set according to those results. (The V and H flags may be altered by these operations, too, but their settings would not be of value after such procedures.)

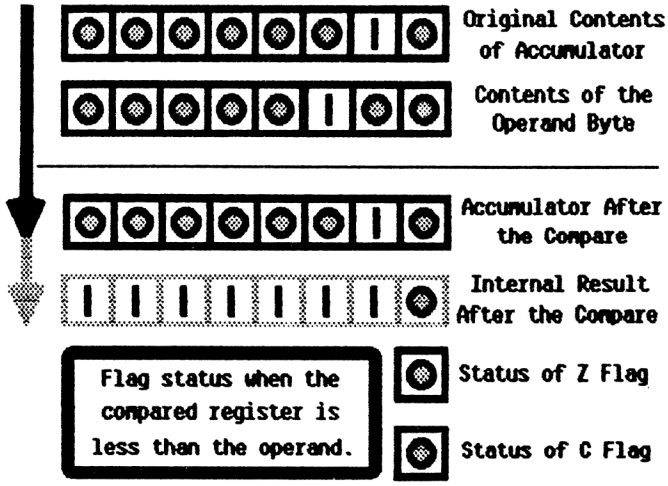
The carry flag is set after the comparison if the result of the comparison indicates that the *operand* was less than the compared value or was equal to the compared value. The carry flag is cleared after the comparison if the operand was greater than the compared value.

The zero flag is set if the comparison indicates that the values were equal. Otherwise it is cleared.

The comparison instructions are among the most important directives in the entire command set in one particular regard: decision making. These instructions enable the computer to make decisions based on the results of comparisons. Other directives, to be introduced shortly, can then


Original Contents of Accumulator
Contents of the Operand Byte
Accumulator After the Compare
Internal Result After the Compare
Flag status when the compared register is larger than the operand.
Status of Z Flag
Status of C Flag


Original Contents of Accumulator
Contents of the Operand Byte
Accumulator After the Compare
Internal Result After the Compare
Flag status when the compared register is equal to the operand.
Status of Z Flag
Status of C Flag


Original Contents of Accumulator
Contents of the Operand Byte
Accumulator After the Compare
Internal Result After the Compare
Flag status when the compared register is less than the operand.
Status of Z Flag
Status of C Flag

cause the computer to execute a different series of instructions (thus modifying its behavior) based on the decisions made as a result of comparisons.

Since these instructions are so central to the productive use of a computer, there is a good complement representing various addressing modes. The first group to be presented compares the contents of the accumulator to locations in memory or to other CPU registers:

Mnemonic	Code
CPA #nn	B7
CPA (U)	27
CPA (X)	07
CPA (Y)	17
CPA nnnn	A7
CPA UH	A6
CPA UL	26
CPA XH	86
CPA XL	06
CPA YH	96
CPA YL	16
CPAI (X)	F7

The last instruction in this group is another

directive that performs automatic incrementing of a data pointer register. It first compares the contents of the accumulator against the location in memory that is *pointed to* by the contents of the 16-bit X register. The address in the X register is then advanced by a count of one. This capability is highly useful when it is necessary to scan a portion of memory while searching for a particular byte of data.

Another group of compare directives provides for the ability to match values in the various CPU registers against immediate data values:

Mnemonic	Code
CPUH #nn	6C
CPUL #nn	6E
CPXH #nn	4C
CPXL #nn	4E
CPYH #nn	5C
CPYL #nn	5E

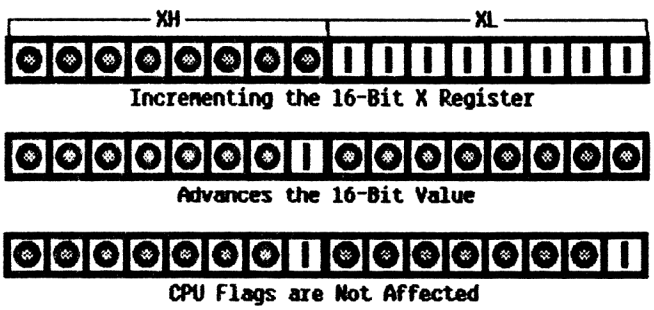
These perform in essentially the same manner as the previous group, with results of the virtual subtraction being used to set up the carry and zero flags. The actual values in the CPU registers (the "compared" values) are not altered.

Increments

The ability to increment the value in a register is important for several reasons. One is so that a value serving as an address pointer can be updated to point to a new location. Another is so that the register can serve as a tally keeper or *counter*. The use of counters is particularly valuable when forming what are known as *program loops*.

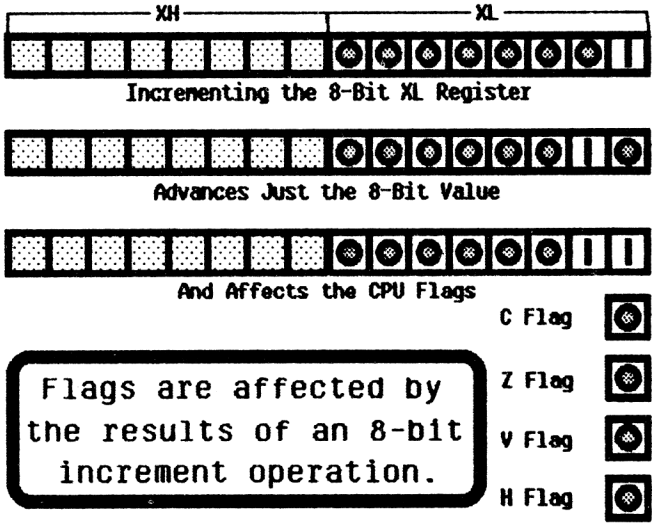
The LH5801 has two basic types of instructions that are used for incrementing values. One type is meant to be used to advance 16-bit data pointer registers. Instructions in this category do not affect the status of the CPU flags.

Mnemonic	Code
INU	64
INX	44
INY	54



The other type is used to advance the count in the accumulator or an 8-bit register (UL, XL or YL). This kind of increment directive will influence the condition of the C, Z, V and H flags.

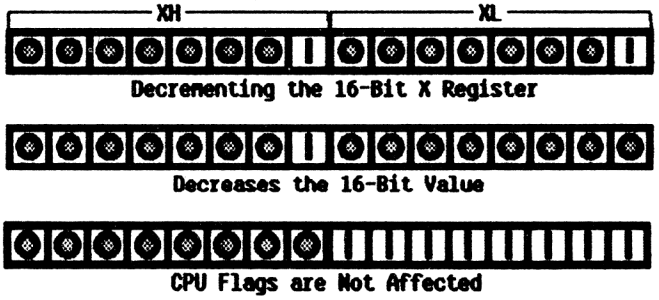
Mnemonic	Code
INA	DD
INUL	60
INXL	40
INYL	50



Decrements

The ability to decrement the values in pointer registers and counters is also valuable. A similar set of directives provides this type of capability in the LH5801. The first subgroup is for decrementing a 16-bit data pointer. None of the flags are affected by the execution of these codes.

Mnemonic	Code
DEU	66
DEX	46
DEY	56

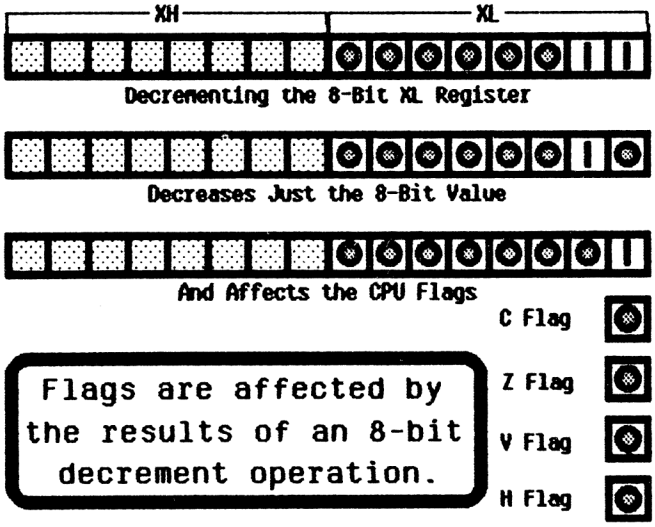


Mnemonic	Code
DEA	DF
DEUL	62
DEXL	42
DEYL	52

The second group decrements an 8-bit register, either the accumulator or UL, XL or YL. Execution of any of these will affect the status of the C, Z, V and H flags. They are set or cleared as appropriate

 Copyright 1983 POCKET COMPUTER NEWSLETTER

according to the result contained in the affected register after the decrement has taken place.



Shifts

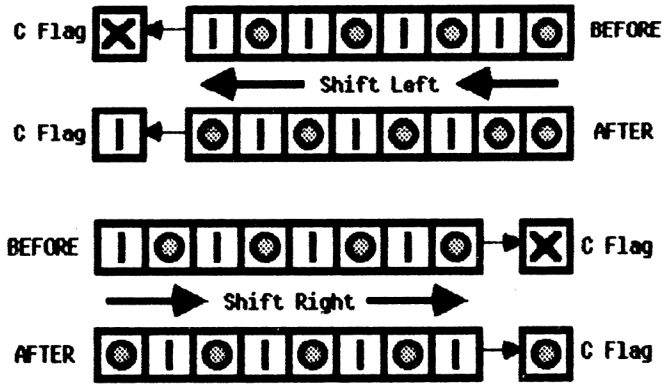
It is often desirable to be able the shift the contents of a register to the right or to the left. The ability to shift data has many uses. Those people who have had experience with digital electronic circuits know this well. Those who have not will soon find it out if they continue to delve into machine language programming.

The LH5801 CPU only has two instructions that are specifically designed for shifting bits. These directives operate directly on the contents of the accumulator. If you ever want to shift the contents of some other CPU register or memory location, you will have to design a little program routine: load the contents of the desired register into the accumulator, shift the contents once they are in the accumulator, then store the result back into the original register!

The two shift directives provided in the LH5801 instruction set are:

Mnemonic	Code
SLA	D9
SRA	D5

The first is used to shift data to the left within the accumulator. The second shifts it to the right.



When a shift occurs, the CPU will always see that the bit at the end of the register (that is being vacated) is cleared to a logic zero regardless of its previous state. On the other end of the register, the bit being kicked out is fed into the carry flag. Thus, you can determine the value of each bit as it is shifted out by examining the status of that (C) flag.

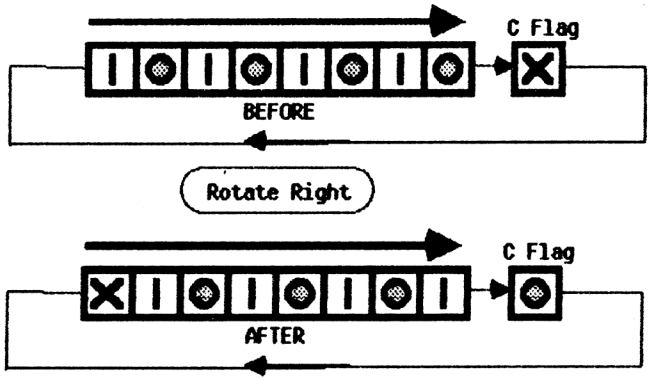
Did you know that shifting a binary value one bit position to the left will multiply it by two? And, yes, shifting it to the right will divide it by two. As you might begin to surmise, the shift operation is a powerful capability to have available when dealing with higher mathematical procedures, such as multiplication and division. It is also handy whenever one wants to deal with data on a bit-by-bit serial basis.

Rotates

If you perform more than one shift operation on the contents of the accumulator, you will lose some data. (Remember, after the first shift, the bit that was shifted out of the register would be temporarily held in the carry flag. As soon as some other instruction that affects the status of the carry flag is executed, including another shift directive, that bit information will be lost.)

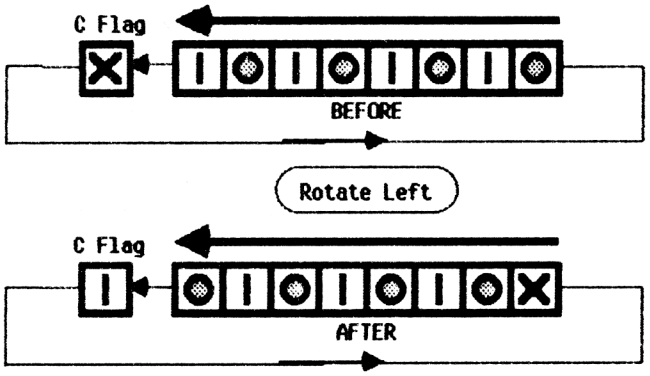
In some situations it may be important to retain all of the information that was originally in the A register, but to reposition that data. Two special directives known as *rotate* commands enable the CPU to perform such a feat.

A register rotate directive is really just a shift with an added feature: the bit shifted out of the register is placed into the carry flag *and* the bit that was originally in the carry flag is fed into the other end of the register. Thus, the bits in the accumulator, coupled by the bit in the carry flag, form a continuous ring.



Rotating to the right results in the carry flag receiving the *least* significant bit from the register. As this is done, the original contents of the carry flag are placed into the *most* significant bit position of the accumulator. The remaining bit positions are all shifted one cell to the right.

Rotating to the left results in the carry flag receiving the *most* significant bit from the register. As this is done, the original contents of the carry flag are placed into the *least* significant bit position of the accumulator. The remaining bit positions are all shifted one cell to the left.



As in the case of the shift directives, there are only two rotate instructions: one to rotate to the left and one to the right. And, the rotating may only be done in the accumulator. You have to bring the contents of other registers into the A register if you want to spin the bits around. Here are the mnemonics and machine codes for the two rotate commands:

Mnemonic	Code
RLCA	DB
RRCA	D1

Please take note of the fact that the initial contents of the carry flag will affect what gets rotated into the accumulator by a rotate directive. You should know what the state of the C flag is before executing such an instruction or else the algorithm being utilized should take account of any possible ambiguity in the state of that flag.

Manipulating the Carry Flag

From time to time I have pointed out that it is often crucial to know the status of the carry flag before executing certain types of instructions. This is particularly true for rotates, and those add and subtract commands that utilize the initial contents of the carry flag.

It is possible to place the carry flag into a known state by executing directives using operands that obtain the desired results. For instance, if the value zero is loaded into a register and then an add-immediate directive is used to add the value zero to that quantity, the carry flag would be cleared. There would be no carry from such an addition operation and the carry flag would reflect that fact. Of course, having to resort to such a procedure every time a programmer wanted to make sure that the carry flag was cleared would be a bit cumbersome. It would also eat up memory as a lot of extra data-containing directives might have to added to a program just to manipulate the flags. On the other hand, it is beneficial to remember that in many programming sequences, it will be possible for the programmer to "know" the status of a flag or "force" various flags to a desired state. In such cases, it is possible to save a byte by not having to call on either of the next two instructions to be presented.

Because the carry flag is so central to many kinds of operations, the instruction set does include two directives that will set it to either possible state: *cleared* (to the logic 0 state) or *set* (to a logic 1 condition).

Mnemonic	Code
CLRC	F9
SETC	FB

Whew! Doesn't that take a load off your mind?

Time to Review

So how do you feel? Do you realize that you have already learned about more than *one hundred and thirty-five* instructions? I hope you feel at least a bit proud!

At this point you should spend some time going over what you have learned. Try doing some actual experimenting with the various commands. You can use POKE directives to tuck small routines into memory. If you want to execute those routines from BASIC using a CALL statement, then be sure and terminate the instruction sequence with the code 9A. This will direct the CPU to go back to the BASIC interpreter when it has finished executing the machine language routine.

In the next installment you will learn about the classes of instructions that provide control and decision-making capability to a computer. Indeed, you will be introduced to much of the rest of the instruction set used in the PC-1500 and PC-2.

After that it will be on to presentations of how various instruction sequences may be combined to perform practical functions. We will start building up a library of *utility* routines that you can call upon. You will also learn how to adapt those routines to serve your own special purposes.

You are well on your way to harnessing the *real* power of your pocket computer!

P.O. Box 232, Seymour, CT 06483

 Copyright 1983 POCKET COMPUTER NEWSLETTER